

ОЛЕКСІЙ ВАСИЛЬЄВ

ПРОГРАМУВАННЯ МОВОЮ PYTHON



ЕЛЕКТРОННИЙ
ВАРІАНТ
КНИГИ



О.М. Васильев

ПРОГРАМУВАННЯ МОВОЮ PYTHON



УДК 004.424
ББК 32.973-018.1
B19

Васильев О.М.

B19 Програмування мовою Python / О.М. Васильев. — Тернопіль: Навчальна книга – Богдан, 2019. — 504 с.; іл.

ISBN 978-966-5611-3

Книга присвячена мові програмування Python. На сьогодні це одна з найпопулярніших та найперспективніших мов. Вона проста, ефективна і красива. Книга допоможе читачеві познайомитися з мовою Python і навчитися використовувати її для написання професійних програм.

Книга містить інформацію про особливості мови Python та необхідне програмне забезпечення. Детально обговорюються основні програмні конструкції, які використовуються при написанні програм. Описуються базові оператори та керуючі інструкції, функції, списки, кортежі, множини, словники і текст. Okрема увага приділяється реалізації в Python парадигми об'єктно-орієнтованого програмування. Теоретичний матеріал ілюструється великою кількістю практичних прикладів.

Книга буде корисною для студентів, слухачів курсів, інженерів, викладачів та всіх, хто вивчає мову програмування Python.

ББК 32.973-018.1

Охороняється законом про авторське право.

*Жодна частина цього видання не може бути використана чи відтворена
в будь-якому вигляді без дозволу видавництва*

Зміст

Вступ

.....	5
Знайомство з мовою Python	6
Коротка історія та особливості мови Python	9
Дещо про книгу	16
Програмне забезпечення	18
Робота з середовищем PyScripter	30
Подяка	37
Зворотний зв'язок	38

Розділ 1. Програма мовою Python..... 39

Розмірковуючи про програму	41
Приклад простої програми	44
Обговорюємо змінні	50
Основні оператори	56
Числові дані	73
Підключення модулів	80
Тернарний оператор	83
Резюме	86

Розділ 2. Інструкції керування..... 89

Умовний оператор	91
Оператор циклу while	102
Оператор циклу for	113
Обробка виняткових ситуацій	125
Резюме	137

Розділ 3. Функції 139

Створення функції	141
Функції для математичних обчислень	145
Значення аргументів за замовчуванням	149
Функція як аргумент	154
Рекурсія	165
Лямбда-функції	170
Локальні та глобальні змінні	176
Вкладені функції	181
Функція як результат функції	184
Резюме	195

Phyton	
--------------	--

Розділ 4. Робота зі списками і кортежами ... 197

Знайомство зі списками	199
Основні операції зі списками	208
Копіювання і присвоювання списків.....	219
Списки та функції	227
Вкладені списки.....	234
Знайомство з кортежами.....	243
Резюме.....	247

Розділ 5. Множини, словники і текст 249

Множини.....	251
Словники	271
Текстові рядки.....	282
Резюме.....	298

Розділ 6. Основи об'єктно-орієнтованого програмування..... 301

Класи, об'єкти й екземпляри класів	303
Конструктор і деструктор екземпляра класу	314
Поле об'єкта класу	320
Додавання й видалення полів	329
Методи і функції	334
Копіювання екземплярів і конструктор створення копії...347	
Резюме.....	359

Розділ 7. Продовжуємо знайомство з ООП 361

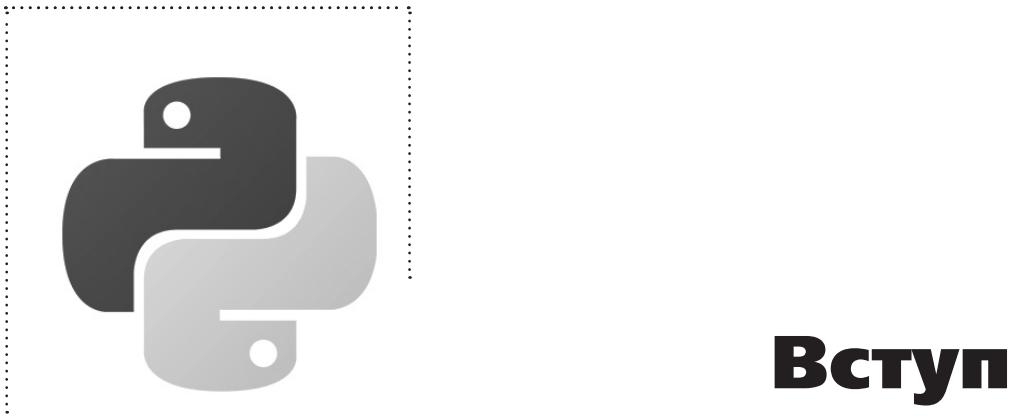
Наслідування	363
Спеціальні методи і поля.....	381
Перевантаження операторів	412
Резюме.....	432

Розділ 8. Коротко про різне 433

Функції зі змінною кількістю аргументів	435
Декоратори функцій і класів.....	445
Документування й анотації у функціях	456
Винятки як екземпляри класів.....	460
Ітератори і функції-генератори	481
Резюме.....	495

Післямова. Про що ми не поговорили..... 497

Запитання і відповіді 502



Вступ

Знайомство з мовою Python

У всьому є своя мораль,
треба лише вміти її знайти!

Л. Керрол. "Аліса в Країні Див"^{*}

У цій книзі йтиметься про те, як писати програми на мові програмування, яка називається Python (правильно читається як пайтон, але зазвичай назву мови читають як пітон, що теж цілком припустимо). Отже, розв'язувати будемо два завдання, одне з яких пріоритетне, а друге, хоч і допоміжне, проте досить важливе. Наше основне завдання — звичайно ж, вивчення синтаксису мови програмування Python. Паралельно ми будемо освоювати ази програмування, явно чи неявно беручи до уваги, що відповідні алгоритми передбачається реалізовувати мовою Python.



Навіть якщо у читача є досвід програмування іншими мовами, не варто ставитися поверхнево чи зверхнью до процесу побудови алгоритму програми. Правила хорошого тону в програмуванні передбачають, що написання програми починається задовго до набирання програмного коду. Непогано взяти аркуш паперу й накреслити загальну схему виконання програми. А для цього процедуру розв'язання великої та складної задачі варто розбити на послідовність простих дій. З одного боку, цей процес універсальний. З іншого — ті задачі, які ми назвали вище “простими”, розв'язуються за допомогою базових команд або функцій мови програмування, якою збираються складати програму. Тому, обмірковуючи алгоритм, цілком абстрагувається від конкретних можливостей мови програмування навряд чи вийде. Ураховуючи ж гнучкість й ефективність мови програмування Python, слід визнати, що алгоритми навіть “класичних” задач, реалізуючись на Python, стають простішими й зрозумілішими. Іншими словами, навіть якщо читач має досвід

* Тут і далі епіграфи подаються у перекладі автора.

складання алгоритмів, знайомство з мовою програмування Python дозволить йому побачити багато знайомих речей зовсім по-іншому.

Узагалі, мов програмування доволі багато. Більше того, час від часу з'являються нові. Тому природно виникає запитання: чому саме Python? Наша відповідь буде складатися з декількох пунктів.

- Мова програмування Python — мова високого рівня, досить “модна”, проте дуже популярна, яка вже зараз широко використовується на практиці, і сфера застосування Python постійно розширюється.



Щодо мов програмування нерідко застосовують такі вирази, як мова **високого рівня**, мова **середнього рівня** або мова **низького рівня**. Ця класифікація досить умовна й базується на рівні абстракції мови. Адже мова, якою розмовляють люди, дещо відрізняється від тієї “мови”, яку “розуміють” комп’ютери. Команда, написана простою людською мовою, буде зовсім неприйнятною для комп’ютера. Команда, готова до виконання комп’ютером (**машинний код**), буде незрозумілою для більшості простих смертних. Тому вибирати доводиться між Сциллою та Харібдою. Про мови, орієнтовані на програміста, говорять, що ці мови високого рівня. Про мови, орієнтовані на комп’ютер, говорять, що ці мови низького рівня. Проміжна група мов називається мовами середнього рівня.Хоча ще раз підкреслимо, що поділ цей досить умовний.

- Синтаксис мови Python мінімалістичний і гнучкий. Цією мовою можна складати прості й ефективні програми.
- Стандартна бібліотека для цієї мови містить багато корисних функцій, що значно полегшує процес створення програмних кодів.
- Мова Python підтримує декілька парадигм програмування, включаючи *структурне*, *об’єктно-орієнтоване* й *функціональне* програмування. І це далеко не весь список.
- Мова Python — цілком вдалий вибір, як для першої мови в навчанні програмуванню.

Існують й інші причини, щоб вивчити мову програмування Python, можливо, навіть вагоміші за перераховані вище. Про деякі ми ще будемо говорити (у контексті особливостей мови програмування Python). У всякому разі, тут будемо виходити з того, що читач для себе ухвалив

рішення про вивчення мови Python або, принаймні, цікавиться цією мовою програмування.



Парадигма програмування – це найзагальніша концепція, яка визначає фундаментальні характеристики й базові методи реалізації програмних кодів. Наприклад, парадигма **об'єктно-орієнтованого** програмування (скоро-чено ООП) передбачає, що програму реалізують через набір взаємодіючих об'єктів, які, у свою чергу, зазвичай створюються на основі класів. У рамках **структурного** програмування програма є комбінацією даних і процедур (функцій) для їх обробки. Мова може підтримувати одразу декілька парадигм. Так, мови Java і C# – повністю об'єктно-орієнтовані, тому для написання найменшої програми цими мовами доведеться описати, як мінімум, один клас. У мові С підтримується парадигма структурного програмування, тому класів й об'єктів у мові С немає. Зате вони є в мові С++. Остання підтримує як парадигму об'єктно-орієнтованого програмування, так і парадигму структурного програмування. Як наслідок, під час роботи з мовою С++ класи й об'єкти можна використовувати, а можна й не використовувати залежно від потреб програміста й специфіки задачі, яку розв'язують. Це ж зауваження стосується й мови Python: із одного боку, під час написання програми мовою Python у нас є можливість удастися до потужного арсеналу об'єктно-орієнтованого програмування, а з іншого боку, часто бувають прийнятними й методи структурного програмування.

Існують й інші, більш витончені концепції програмування. Скажімо, парадигма функціонального програмування припускає, що результат функції в програмі визначається виключно значеннями аргументів, переданих функції, і не залежить від стану зовнішніх (стосовно функції) змінних. Відповідні функції прийнято називати **чистими функціями**, і вони мають ряд корисних властивостей, що дозволяють істотно оптимізувати й прискорити обчислювальний процес. Ця концепція, як і ряд інших, знаходить реалізацію в мові Python.

Далі обговоримо деякі важливі моменти й “підводне каміння”, яке може зустрітися на подекуди важкому, та все ж цікавому й захопливому шляху опанування новими вершинами у програмуванні.

Коротка історія та особливості мови Python

Серйозне ставлення до будь-чого в цьому світі є фатальною помилкою.

Л. Керром. “Аліса в Країні Див”

У мови Python є автор *Гвідо ван Россум* (Guido van Rossum). І хоча в розробці й популяризації мови на теперішній момент устигло взяти участь багато талановитих розробників, саме Гвідо ван Россум отримав заслужену славу творця цієї перспективної та популярної мови програмування. Взагалі ж робота над мовою почалася у 80-х роках минулого століття. Вважають, що перша версія мови з'явилася в 1991 році. Щодо назви мови програмування Python, то, формально, це назва рептилії. Відповідно, часто як логотип використовують милу (або не дуже) змійку типу “пітон”. І хоча практично будь-який навчальний чи довідковий посібник із мови Python містить розповідь про те, що насправді Python це не “пітон”, а назва гумористичної передачі “Літаючий цирк Монті Пайтона”, для історії це вже не важливо.



Доречно згадати слова капітана Брунгеля з однійменної повісті Андрія Некрасова: “Як ви яхту назовете, так вона й попливе”. У мови програмування Python досить агресивна назва, і, треба визнати, ця назва себе виправдовує. За аргумент до такого твердження може правити як гнучкість та ефективність самої мови, так і та швидкість, із якою вона завоювала собі “місце під сонцем” серед найпопулярніших мов програмування.

Мова Python бурхливо розвивається. Цьому сприяє не тільки досить вдала концепція мови, а також згуртоване співтовариство, сформоване

з розробників і популяризаторів мови. Важливий і той факт, що необхідне програмне забезпечення, включаючи середовища розробки, в основному безкоштовне. Усе це дає підстави розглядати Python як одну з найперспективніших мов програмування.

На сьогодні Python використовується при реалізації найрізноманітніших проектів, серед яких:

- розробка сценаріїв для роботи з Web та Internet-програмами;
- мережеве програмування;
- засоби підтримки технологій HTML і XML;
- програми для роботи з електронною поштою й підтримки Internet-протоколів;
- програми для обслуговування найрізноманітніших баз даних;
- програми для наукових розрахунків;
- програми з графічним інтерфейсом;
- створення ігор і комп’ютерної графіки та багато іншого.

Зрозуміло, охопити всі ці теми в одній книзі досить складно. Та ми й не ставимо це собі за мету. А втім, навіть на відносно невеликій кількості нескладних прикладів цілком можливо продемонструвати елегантність і виключну ефективність мови Python. Цим, власне, ми й займемося в основній частині книги — тобто трохи згодом. Зараз обговоримо особливості та деякі «технічні» моменти, які важливі для розуміння основ програмування мовою Python.

Python належить до мов програмування, *що інтерпретуються*, і це має певні наслідки. Формально те, що мова програмування належить до інтерпретованих, означає, що програмний код виконується за допомогою спеціальної програми-інтерпретатора. Інтерпретатор виконує програмний код порядково (з попереднім аналізом виконуваного коду). Недолік такого підходу полягає в тому, що, по-перше, помилки виявляються фактично на етапі виконання програми і, по-друге, швидкість виконання програми відносно невисока. Тому нерідко застосовується більш складна схема: вихідний програмний код компілюється в проміжний код, а вже цей проміжний код виконується безпосередньо інтерпретатором. У цьому разі швидкість виконання програми збільшується, але разом із нею збільшується і потреба у системних ресурсах. Приблизно за такою схемою виконується програмний код, написаний мовою Python.



Нагадаємо, що окрім мов, які інтерпретуються, існують мови, програми на яких **компілюються** (мається на увазі компіляція в машинний код). У цьому випадку вихідний програмний код компілюється у виконавчий (машинний) код, який виконується (зазвичай) під керуванням операційної системи. Компільовані у виконавчий код програми характеризуються відносно високою швидкістю виконання.

Якщо ми хочемо написати програму на мові Python, то для цього, як мінімум, знадобиться набрати відповідний програмний код. Тут можливі варіанти, але, в принципі, код набирається в будь-якому текстовому редакторі, а відповідний файл зберігається з розширенням `py` чи `pyw` (для програм із графічним інтерфейсом). Під час першого запуску (після внесення змін у програмний код) створюється проміжний код, який записується у файл із розширенням `pyc`. Якщо після цього в програму зміни не вносилися, то під час виконання програми буде виконуватися відповідний `pyc`-файл. Після внесення змін у програму під час чергового запуску вона перекомпілюється в `pyc`-файл. Це загальна схема. Нас, насправді, вона цікавить виключно в плані загального розвитку, хоча на практиці іноді буває важливо враховувати особливості виконання програми, написаної мовою Python.

Як зазначалося вище, програмний код можна набирати й у текстовому редакторі. Та ось для виконання такого програмного коду знадобиться спеціальна програма, яка називається *інтерпретатором*. Іншими словами, для роботи з Python на комп'ютер необхідно встановити програму-інтерпретатор. Ми окремо зупинимося на цьому питанні. Зараз же тільки зауважимо, що зазвичай використовується не просто інтерпретатор, а *інтегроване середовище розробки*, яке, крім іншого, включає в себе як інтерпретатор, так і редактор програмних кодів.

Інтерпретатор виконує програму команда за командою. Тому, в принципі, якщо програма складається з декількох команд, її можна організувати у вигляді файлу з програмним кодом, а потім «відправити» цей файл на виконання. Ще один варіант — «передавати» інтерпретатору для виконання по одній команді. Обидва режими можливі й підтримуються інтерпретатором мови Python. Ми будемо складати і запам'ятовувати програмні коди у файлах — тобто використаємо «традиційний» підхід.



Про режим, за якого в командному вікні інтерпретатора команди вводяться і виконуються одна за одною, говорять, як про **режим командного рядка**, або **режим калькулятора**.

Оскільки мова Python розвивається інтенсивно, їй від версії до версії в синтаксис і структуру мови вносяться зміни, важливо враховувати, для якої версії мови Python складають (і передбачають використовувати) програмний код. Особливо це важливо, враховуючи ту обставину, що під час внесення змін *принцип зворотної сумісності* спрацьовує далеко не завжди: якщо програмний код коректно виконується в давніших версіях мови, то зовсім не факт, що він буде виконуватися під час роботи з новішими версіями. Однак панікувати з цього приводу не варто. Зазвичай проблема несумісності кодів для різних версій пов'язана з особливостями синтаксису деяких функцій або конструкцій мови і досить легко усувається.



На момент написання книги актуальною є версія Python 3.6. Саме вона використовувалася для тестування прикладів у книзі. У разі виходу нових версій або стандартів мови, для забезпечення сумісності програмних кодів є сенс проглянути той розділ довідкової системи, в якому описано нововведення. Зробити це можна, наприклад, на офіційному сайті підтримки мови Python www.python.org/doc/ у розділі під назвою *What's New In Python* (у перекладі означає *Що нового в Python*).

Але все це технічні деталі, які хоч і важливі, та все ж не першорядні. А першорядними для нас будуть синтаксис мови Python і його основні інструкції керування. І, власне, тут на нас чекає чимало приемних сюрпризів.



Особливо багато сюрпризів буде для читачів, які знайомі з такими мовами програмування, як Java чи C++, наприклад. Але це, до речі, зовсім не означає, що новачків у програмуванні мова Python залишить байдужими. Просто ті, хто вивчав основи ООП і програмує згаданими мовами, значно розширяє свій кругозір щодо методів і прийомів програмування, а також, у певному сенсі, їм доведеться змінити своє уявлення про мови програмування.

Синтаксис мови Python більш ніж цікавий. По-перше, він простий, зрозумілий і наочний. Його навіть можна назвати по-спартанськи лаконічним.

Разом з цим програмні коди, написані на Python, зазвичай легко читаються й аналізуються, а обсяг програмного коду набагато менший, порівняно з аналогічними програмами, написаними іншими мовами програмування. Як ілюстрацію в листингу В.1 наведено код програми мовою C++, у результаті виконання якого в консольне вікно виводиться повідомлення Hello, world!.

Листинг В.1. Програма мовою C++

```
#include <iostream>
using namespace std;
int main() {
    cout<<"Hello, world!"<<endl;
    return 0;
}
```

Аналогічний програмний код, але вже мовою Java, представлено в листингу В.2.

Листинг В.2. Програма мовою Java

```
class MyClass{
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Нарешті, у листингу В.3 показано, як виглядатиме програма для виведення в консольне вікно текстового повідомлення, якщо для її написання скористатися мовою програмування Python.

Листинг В.3. Програма мовою Python

```
print("Hello, world!")
```

Неважко помітити, що це всього одна команда, де вбудованій функції print() як аргумент передається текст, який необхідно надрукувати в консольному вікні. Зрозуміло, далеко не завжди у нас буде виходити писати такі «економні» коди, але приклад усе ж багато в чому показовий.



Про всякий випадок, коротко прокоментуємо наведені вище коди мовами C++ і Java — просто, щоб у читача, не знайомого з цими мовами, не виникло комплексу меншовартості. Почнімо з програмою лістингу В.1, написаної мовою C++:

- Інструкція `#include <iostream>` підключає заголовковий файл бібліотеки вводу/виводу.
- Команда `using namespace std` означає використання стандартного простору імен.
- Функція з назвою `main()` називається головною функцією програми. Виконання програми в C++ — це виконання головної функції програми.
- Ідентифікатор `int` ліворуч від функції `main()` означає, що функція повертає цілочисловий результат.
- Пара фігурних дужок (`{ i }`) виділяє тіло головної функції.
- Команда `cout << "Hello, world!" << endl` виводить у консоль текстове повідомлення `Hello, world!`, і відбувається перехід до нового рядка (інструкція `endl`). Оператор виводу `<<` виводить текст, зазначений праворуч від нього, на пристрій, визначений ідентифікатором `cout` (за замовчуванням — консоль).
- Інструкція `return 0` завершує виконання головної функції (тобто програми), а результатом функція повертає 0 (означає закінчення роботи програми «у штатному режимі» — тобто без помилок).

Програма в лістингу В.2, нагадаємо, написана мовою Java, і у відповідному програмному коді призначення інструкцій таке:

- Створюється клас із назвою `MyClass`: перед назвою класу зазначається ключове слово `class`, а тіло класу береться у фігурні дужки (зовнішня пара дужок `{ i }`).
- У тілі класу описується головний метод із назвою `main()`, тіло методу береться в блок із фігурних дужок (внутрішня пара дужок `{ i }`).
- Перед назвою головного методу зазначено такі ідентифікатори: `public` (відкритий метод — тобто доступний поза класом), `static` (статичний метод — для виклику методу немає необхідності створювати об'єкт класу), `void` (метод не повертає результат).
- Як параметр (аргумент) методу `main()` указано змінну `args`, яка є текстовим масивом (текст — це об'єкт класу `String`, а наявність порожніх квадратних дужок `[]` свідчить про те, що це текстовий масив — упорядкований набір текстових значень).
- Текст у консольне вікно виводиться за допомогою методу `println()`: як аргумент методу передається текст, що виводиться в консоль, а сам метод викликається з об'єкта потоку виводу `out`, який, у свою чергу, є статичним полем класу `System`.

Зрозуміло, на фоні всього цього різноманіття програма (а точніше, одна єдина команда) мовою Python виглядає більш ніж ефектно. Але ще раз підкреслюємо, що, навіть якщо читач зрозумів не все з викладеного вище (щодо кодів C++ і Java), це не страшно. Нам, у нашій подальшій роботі, все це не знадобиться. Ми просто хотіли проілюструвати масштаби, так би мовити, розбіжностей для різних мов.

Під час роботи з мовою Python ми не зустрінемо багатьох звичних для інших мов конструкцій. Наприклад, на відміну від мов C++ і Java, в яких командні блоки виділяються фігурними дужками { і }, і на відміну від мови Pascal, у якій блоки виділяються за допомогою інструкцій begin і end, у мові Python блок команд виділяється відступом (рекомендованій відступ — чотири пробіли). У мові Python немає необхідності закінчувати кожну команду крапкою з комою. Існують й інші особливості мови Python. Ми будемо знайомитися з ними поступово і, перефразовуючи М.Є. Салтикова-Щедріна, російського письменника-сатирика, намагатимемося при цьому не застосовувати силу.



Мається на увазі цитата «Просвіту впроваджувати помірковано, по можливості уникаючи кровопролиття» із сатиричного роману «Історія одного міста» М.Є. Салтикова-Щедріна.

Тут просто важливо зрозуміти, що мова Python досить своєрідна, самобутня й у багатьох відношеннях не схожа на інші популярні сьогодні мови програмування.

Дещо про книгу

Вона завжди давала собі хороші поради, хоча слідувала їм нечасто.

Л. Керром. «Аліса в Країні Див»

Настав час сказати (прочитати, написати — кому як більше подобається) декілька слів безпосередньо про книгу: що вона собою являє, для кого написана, ѹ, узагалі, що читачеві очікувати від прочитання.

Зрозуміло, книга писалася, насамперед, для тих, хто вирішив опанувати мову програмування Python. Тобто передбачається, що читач із мовою Python не знайомий узагалі. Більше того, ми неявно будемо виходити з того, що читач має, загалом, мінімальний досвід програмування. Останнє не завадить нам періодично посилатися до таких мов програмування, як C++ і Java. Звичайно, робитимемо ми це, передусім, розраховуючи на тих читачів, хто має хоча б мінімальну уяву про ці мови і/або ООП. Щоб компенсувати незручності, які могли б виникнути у читачів, не знайомих із C++ і Java, пояснення максимально адаптуються для сприйняття повністю непідготовленою аудиторією. Простіше кажучи, не має значення, знає читач інші мови програмування чи ні — у будь-якому разі він може розраховувати на успіх.

Матеріал книги охоплює всі основні теми, необхідні для успішної роботи з мовою Python, включаючи методи ООП. В основному, ми будемо розглядати конкретні задачі — тобто теорію буде наведено «на прикладах». Це прийом, який на практиці непогано себе зарекомендував. Особливо він ефективний, коли необхідно в стислі терміни з мінімальними затратами енергії і ресурсів опанувати на якісному рівні великий обсяг матеріалу. За такого підходу є й додатковий бонус: окрім особливостей мови читач

має можливість познайомитися з алгоритмами, які застосовують для розв'язання ряду прикладних задач. Щодо підбору прикладів і задач, то вони вибиралися так, щоб найяскравіше проілюструвати можливості їх особливості мови Python.

Програмне забезпечення

– А де я можу знайти кого-небудь нормального?
– Ніде, – відповів Кіт, – нормальних не буває. Адже всі такі різні і несхожі. І це, по-моєму, нормальноН.

Л. Керролл. «Аліса в Країні Див»

Перш ніж поринути в глибини світу під назвою Python і почати здобувати безцінні знання, доречно внести ясність у питання про програмне забезпечення, яке нам знадобиться для тестування прикладів із книги і написання власних оригінальних програм.

Як зазначалося вище, для виконання програмних кодів, написаних на Python, нам потрібна програма-інтерпретатор. Але найкраще скористатися будь-яким інтегрованим середовищем розробки (скорочено *IDE* від англійського *Integrated Development Environment*). Середовище розробки надає користувачу не тільки інтерпретатор, а й редактор кодів, так само як і ряд інших корисних утиліт. Інтегрованих середовищ розробки для роботи з Python існує досить багато, і в певному сенсі перед програмістом виникає непроста проблема вибору. Критерії для вибору середовища розробки можуть бути найрізноманітнішими. Але головні серед них, звичайно ж, — це зручність у використанні, набір вбудованих можливостей/функцій інтегрованого середовища розробки, а також його вартість (існують як комерційні продукти, так і у вільному доступі). Ми розглянемо декілька найпопулярніших і найдоступніших інтегрованих середовищ розробки для Python.

Якщо ми говоримо про програмне забезпечення, то, в першу чергу, є сенс вийти на офіційний сайт підтримки Python за адресою www.python.org. Вікно браузера, відкрите на відповідній сторінці, показано на рис. В.1.

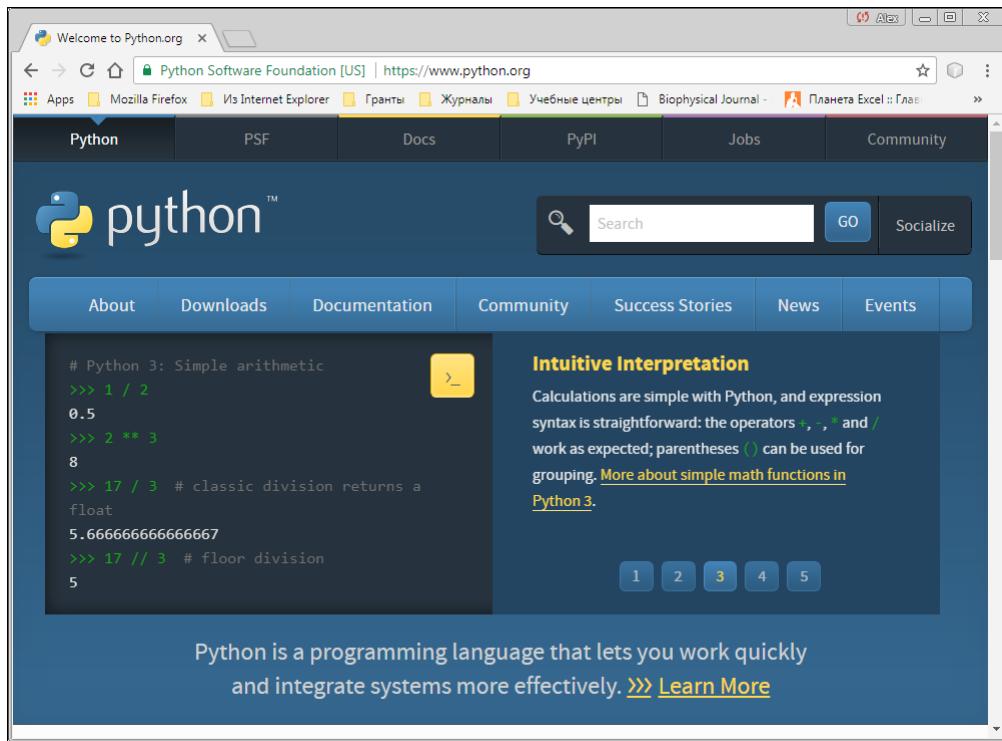


Рис. В.1. Вікно браузера, відкрите на офіційній сторінці підтримки Python
www.python.org

Сайт містить чимало корисної інформації, включаючи всеосяжну довідку, і дозволяє завантажити необхідне програмне забезпечення — у тому числі, й середовище розробки, яке називається *IDLE* (скорочення від *Integrated Development Environment*, що буквально означає *інтегроване середовище розробки*). Для завантаження програмного забезпечення необхідно перейти до розділу *Downloads* (*завантаження*) — адреса www.python.org/downloads.

Сам процес завантаження й установки досить простий та інтуїтивно зрозумілий, тому зупинятися на ньому не будемо. Нас цікавить кінцевий результат. А в результаті ми отримуємо повноцінне середовище для роботи

Phyton

з програмними кодами Python. Робоче вікно середовища IDLE представлено на рис. В.2.

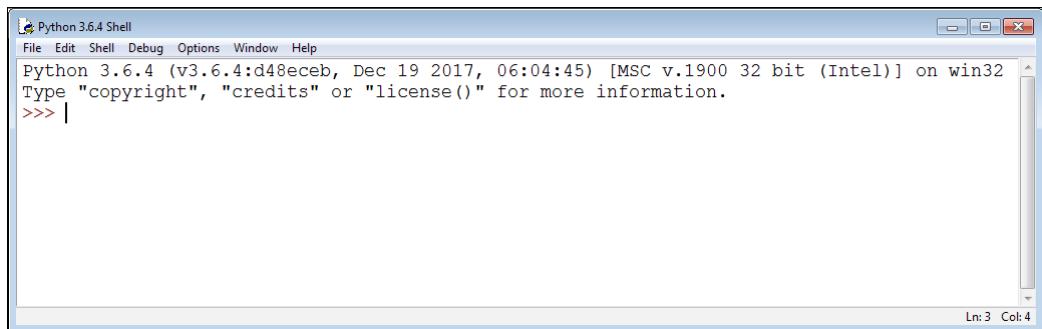


Рис. В.2. Робоче вікно середовища розробки IDLE

Перед нами командна оболонка інтерпретатора. Це вікно з декількома меню і великою робочою областю, в якій після символу потрійної стрілки >>> блимає курсор — це командний рядок. У цьому місці вводиться команда, яка виконується після натискання клавіші <Enter>. Наприклад, якщо ми хочемо виконати команду `print("Hello, world!")`, нам треба ввести цю команду в командний рядок (тобто там, де блимає курсор — після символу >>>) і натиснути клавішу <Enter>. Як наслідок, команду буде виконано, а результат її виконання відобразиться знизу, під командним рядком. Ситуацію проілюстровано на рис. В.3.

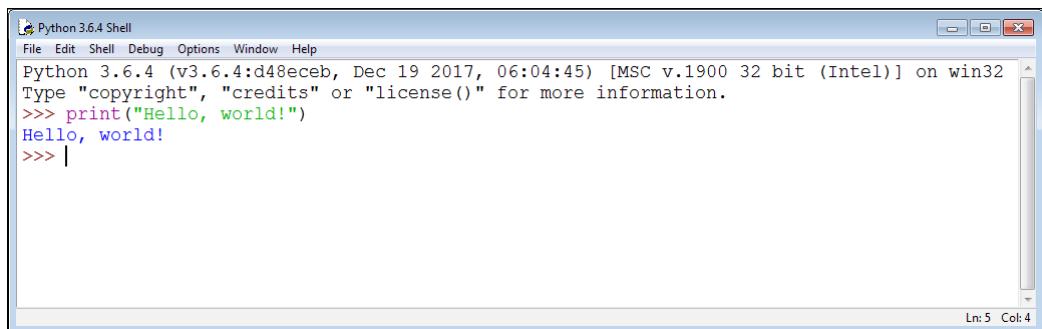


Рис. В.3. Результат виконання команди в командній оболонці середовища розробки IDLE

При цьому під результатом виконання команди знову з'являється потрійна стрілка >>>, і в цьому місці можна вводити нову команду. Наприклад,

можемо ввести який-небудь алгебраїчний вираз — скажімо, нехай це буде $5+3*4$, як показано на рис. В.4.

```
Python 3.6.4 (v3.6.4:d48ebeb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello, world!")
Hello, world!
>>> 5+3*4
17
>>> |
```

Ln: 7 Col: 4

Рис. В.4. Результат обчислення алгебраїчного виразу

Зрозуміло, команди можуть бути й більш хитромудрі, так само як ніхто не забороняє нам команду за командою виконувати програмний код. Але це досить незручно. Зазвичай при написанні більш-менш серйозної програми її оформлюють у вигляді послідовності інструкцій і записують в окремий файл. Потім відповідна програма виконується.

Створити файл програми можна за допомогою все тієї ж оболонки середовища розробки. Якщо клацнути меню File, відкриється список команд і підменю, серед яких є й команда New File (рис. В.5).

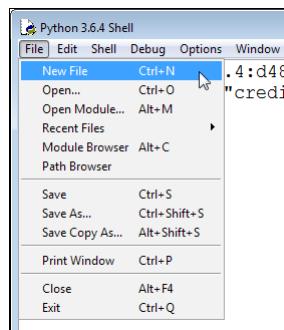


Рис. В.5. Створення файла з програмою

Одразу після вибору цієї команди відкривається редактор кодів, показаний на рис. В.6.

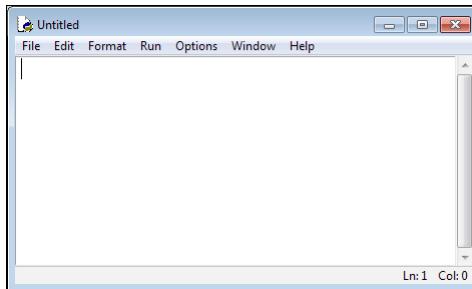


Рис. В.6. Редактор кодів для створення файла з програмою

У вікні редактора вводимо програмний код. У цьому разі наша програма складатиметься лише з декількох команд, наведених у лістингу В.4.

Лістинг В.4. Декілька команд для запису у файл

```
print("Починаємо обчислення!")
a=4
print("Значення змінної a = ",a)
b=12
print("Значення змінної b = ",b)
c=b/a
print("Результат ділення b/a = ",c)
print("Обчислення закінчено!")
```

Саме такий програмний код ми вводимо у вікні редактора кодів. Вікно редактора з програмним кодом показано на рис. В.7.

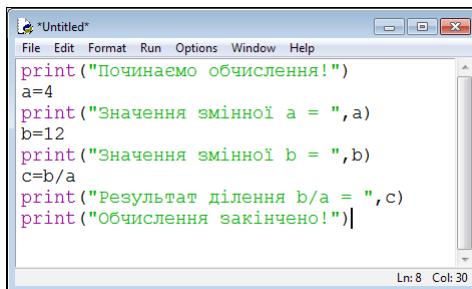
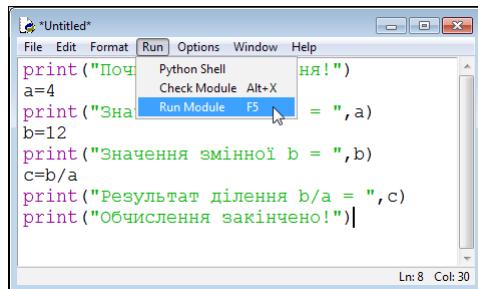


Рис. В.7. Вікно редактора кодів із кодом програми

Після того, як програмний код набрано, його можна одразу виконати. Для цього в меню Run вибираємо команду Run Module, як показано на рис. В.8.



```

print("Поч...")
a=4
print("Зна...")
b=12
print("Значення змінної b = ",b)
c=b/a
print("Результат ділення b/a = ",c)
print("Обчислення закінчено!")

```

Рис. В.8. Запуск програми на виконання

Правда, попередньо все ж таки краще зберегти файл із програмою, для чого корисно буде команда Save із меню File (рис. В.9).

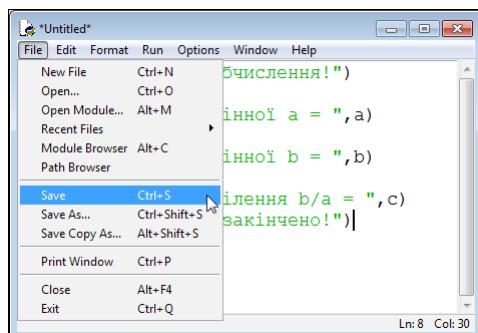


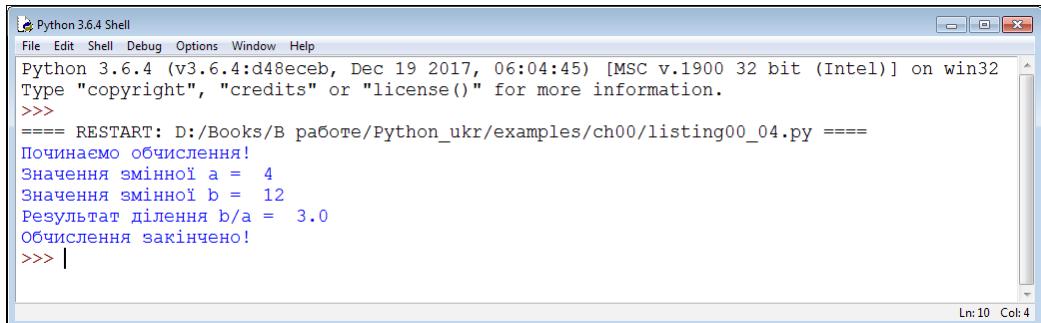
Рис. В.9. Збереження файла з програмою



Якщо перед запуском програми на виконання файл не зберегти, з'явиться діалогове вікно з пропозицією зберегти файл. Тому краще це зробити одразу.

Проте, хоч би там що було, в результаті виконання програми у вікні командної оболонки з'явиться результат, як на рис. В.10.

Phyton



```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Books/В работе/Python_ukr/examples/ch00/listing00_04.py =====
Починаємо обчислення!
Значення змінної a = 4
Значення змінної b = 12
Результат ділення b/a = 3.0
Обчислення закінчено!
>>> |
```

Рис. В.10. Результат виконання програми

Як бачимо, в області виводу результатів (під символом >>>) з'являється декілька повідомлень, які, очевидно, є наслідком виконання програми, наведеної в листингу В.4. Результат програми подано нижче.



Результат виконання програми (з листингу В.4)

Починаємо обчислення!
Значення змінної a = 4
Значення змінної b = 12
Результат ділення b/a = 3.0
Обчислення закінчено!

І хоча ми «офіційно» ще нібіто не розпочали вивчення мови Python, є сенс прокоментувати відповідні команди і результат їхнього виконання.

Отже, команда `print("Починаємо обчислення!")` на початку виконання програми виводить текстове повідомлення `Починаємо обчислення!`. Аналогічно, команда `print("Обчислення закінчено!")` наприкінці програмного коду виводить текстове повідомлення `Обчислення закінчено!`, що свідчить про завершення виконання програми.

Між цими командами виконуються невеликі обчислення:

- за допомогою команди `a=4` змінній a присвоюється числове значення 4;
- за допомогою команди `print("Значення змінної a = ", a)` виводиться текст, а потім значення змінної a;

- за допомогою команди `b=12` змінній `b` присвоюється числове значення `12`;
- за допомогою команди `print("Значення змінної b = ", b)` виводяться текст і значення змінної `b`;
- за допомогою команди `c=b/a` змінній `c` як значення присвоюється результат ділення значення змінної `b` на значення змінної `a`;
- за допомогою команди `print("Результат ділення b/a = ", c)` виводяться текст і числове значення змінної `c`.

У справедливості цих тверджень читач може переконатися, що раз поглянувши на рис. B.10.



Напевно, допитливий читач помітив, що змінні в програмному коді використано без оголошення їхнього типу. Іншими словами, ми ніде явно не зазначали тип змінних, які використовували в програмі. Це стандартна ситуація для програм, написаних на Python, — тип змінних не зазначається (він визначається автоматично за значенням, яке присвоюється змінній).

Також ми побачили, що функції `print()` можна передавати не тільки один, а декілька аргументів. У цьому випадку в область виводу (або консоль) по-слідовно, в один рядок, виводяться значення аргументів функції `print()`.

Про те, як правильно створювати програмні коди на Python, ми будемо говорити в основній частині книги. Тут нам важливо лише проілюструвати, що потім із цими програмними кодами робити. Також нам важливо дати читачеві найзагальніше уявлення про ті прикладні програми і середовища розробки, які дозволяють у зручному режимі створювати коди і запускати їх на виконання. щодо коротко описаного вище середовища IDLE, то назвати його дуже вже вдалим навряд чи можна, хоча, звичайно, це суб'єктивна точка зору автора, і читач не зобов'язаний її поділяти.

Серед комерційних продуктів можна виділити інтегроване середовище розробки *Komodo IDE* (офіційний сайт www.activestate.com). Вікно середовища розробки з відкритим у ньому файлом програми, що розглядалася вище, показано на рис. B.11.

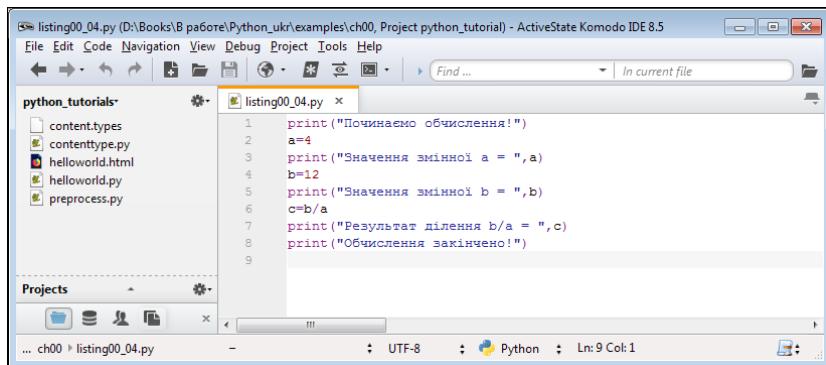


Рис. В.11. Вікно інтегрованого середовища розробки Komodo IDE з програмним кодом

Результат виконання програми в середовищі Komodo IDE показано на рис. В.12 (для запуску програми на виконання можна скористатися командою Run Without Debugging із меню Debug).

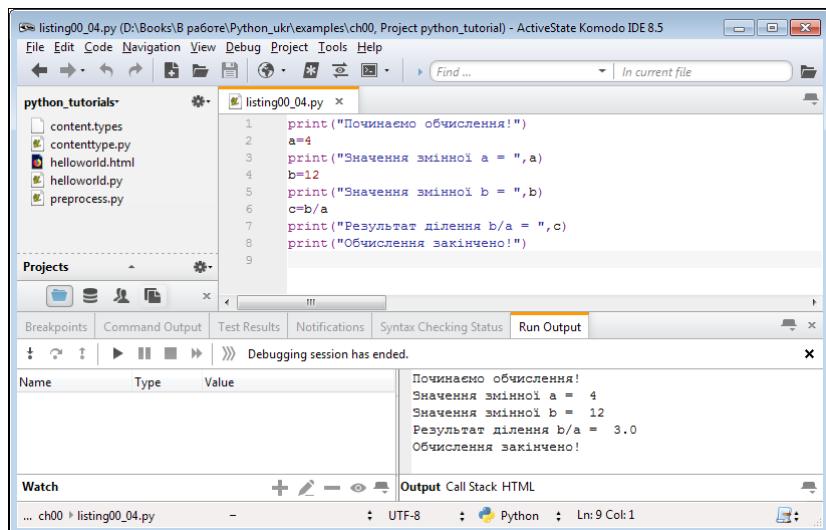


Рис. В.12. Результат виконання програми в середовищі Komodo IDE

Ми, однак, використовуватимемо для тестування прикладів із книги не-комерційне, зручне і просте середовище розробки *PyScripter*. Інсталяційні файли можна вільно завантажити у розділі Downloads на сторінці <https://sourceforge.net/projects/pyscripter>. Сторінку підтримки проекту PyScripter з відкритим вікном браузера показано на рис. В.13.

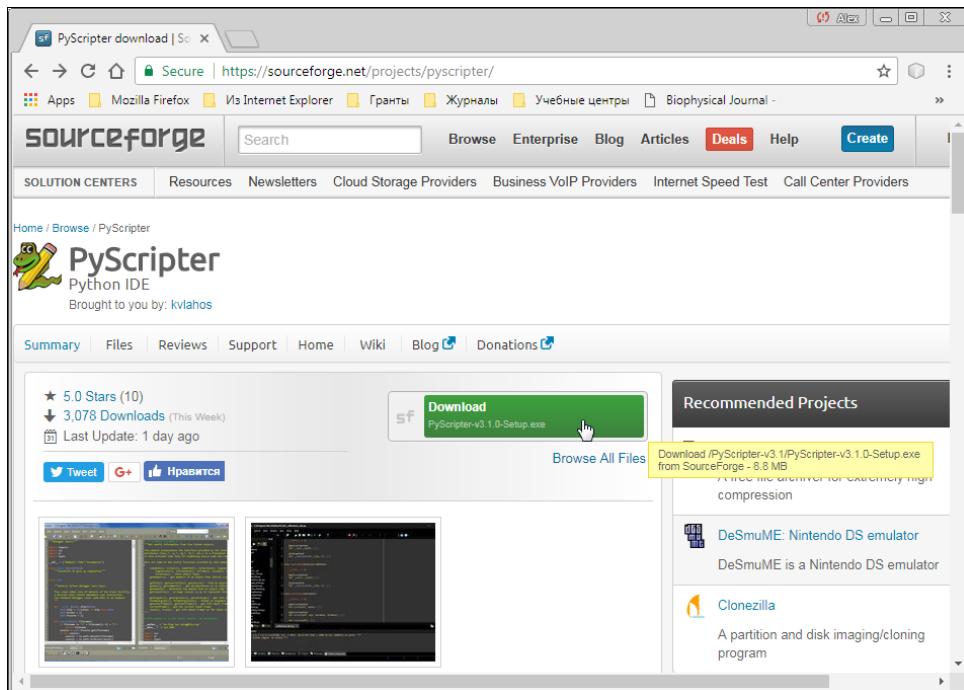


Рис. В.13. Сторінка підтримки проекту PyScripter

Вікно середовища розробки з програмним кодом показано на рис. В.14.

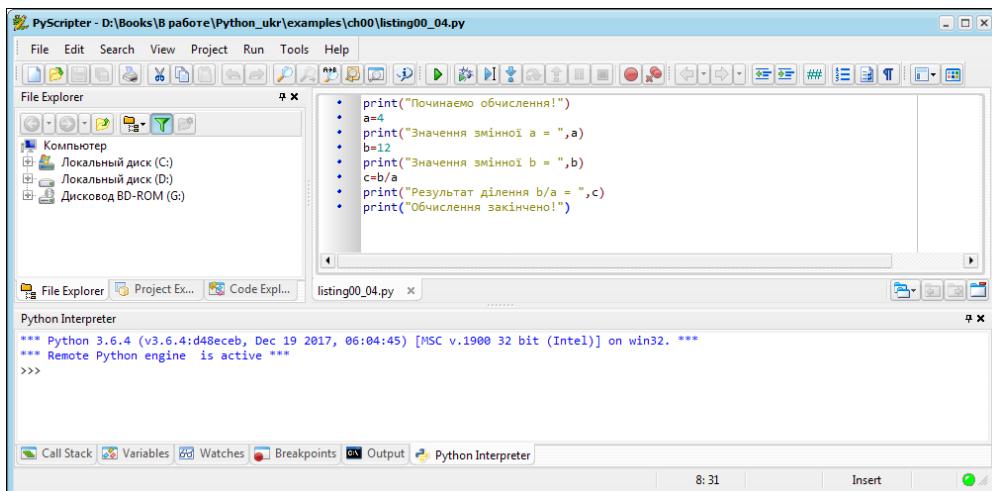


Рис. В.14. Вікно середовища PyScripter із програмним кодом



Щоб створити новий файл із програмою, вибираємо команду **New** в меню **File**, а щоб відкрити вже існуючий файл, вибираємо команду **Open** із того ж меню.

Для запуску програми на виконання вибираємо в меню **Run** команду **Run** (рис. В.15) або натискаємо кнопку з зеленою стрілкою на панелі інструментів.

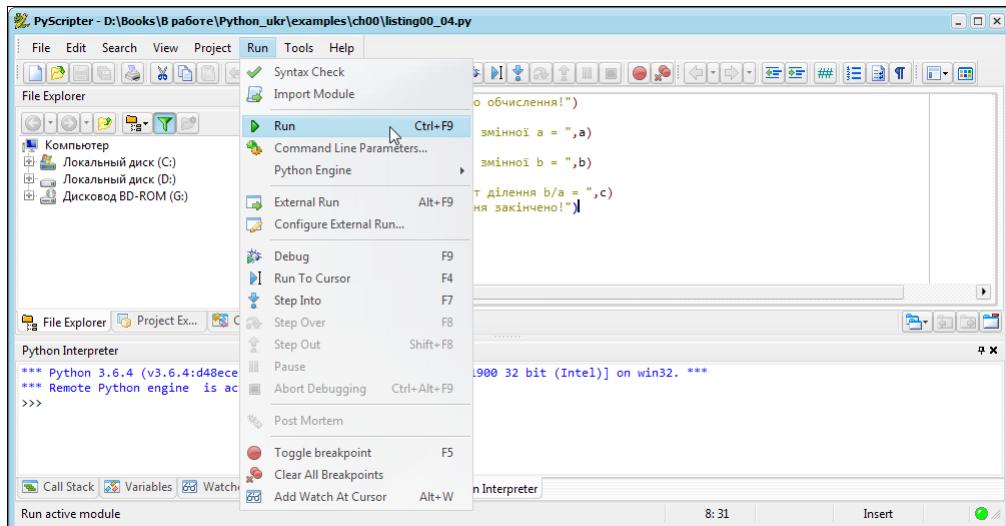


Рис. В.15. Запуск програми на виконання в середовищі PyScripter

На рис. В.16 показано результат виконання програми.

The screenshot shows the PyScripter IDE interface. On the left is the 'File Explorer' pane showing local drives (C:, D:, G:). The main workspace contains the code for listing00_04.py:

```

print("Починаємо обчислення!")
a=4
print("Значення змінної a = ",a)
b=12
print("Значення змінної b = ",b)
c=b/a
print("Результат ділення b/a = ",c)
print("Обчислення закінчено!")

```

Below the code is the 'Python Interpreter' window displaying the execution results:

```

*** Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32. ***
*** Remote Python engine is active ***
>>> *** Remote Interpreter Reinitialized ***
>>>
Починаємо обчислення!
Значення змінної a = 4
Значення змінної b = 12
Результат ділення b/a =  3.0
Обчислення закінчено!
>>>

```

The bottom status bar shows 'Script run OK'.

Рис. В.16. Результат виконання програми в середовищі PyScripter

Результат відображується у внутрішньому вікні Python Interpreter, і це доволі зручно.



За бажанням у внутрішньому вікні Python Interpreter можна виконувати окремі команди: інструкції для виконання вводяться після символу >>>.

Оскільки в наші плани входить широке використання середовища розробки PyScripter для роботи з програмними кодами, далі ми більш детально обговоримо деякі особливості цієї програми. Також зазначимо, що якщо читач унаслідок якихось об'ективних або суб'ективних причин надасть перевагу іншому середовищу розробки (у тому числі й одному з перерахованих вище) — немає жодних проблем. Правда, на сторінках книги відсутня можливість описати всі (або навіть деякі) найпопулярніші середовища розробки для Python: книга, все-таки, присвячена мові програмування, а не програмному забезпеченню. Та й більшість пропонованих утиліт для роботи з програмними кодами Python зазвичай прості у використанні, універсальні в плані методів і прийомів роботи з ними, а також інтуїтивно зрозумілі. Хочеться вірити, що читач у разі потреби сам зможе впоратися з опануванням необхідного програмного забезпечення.

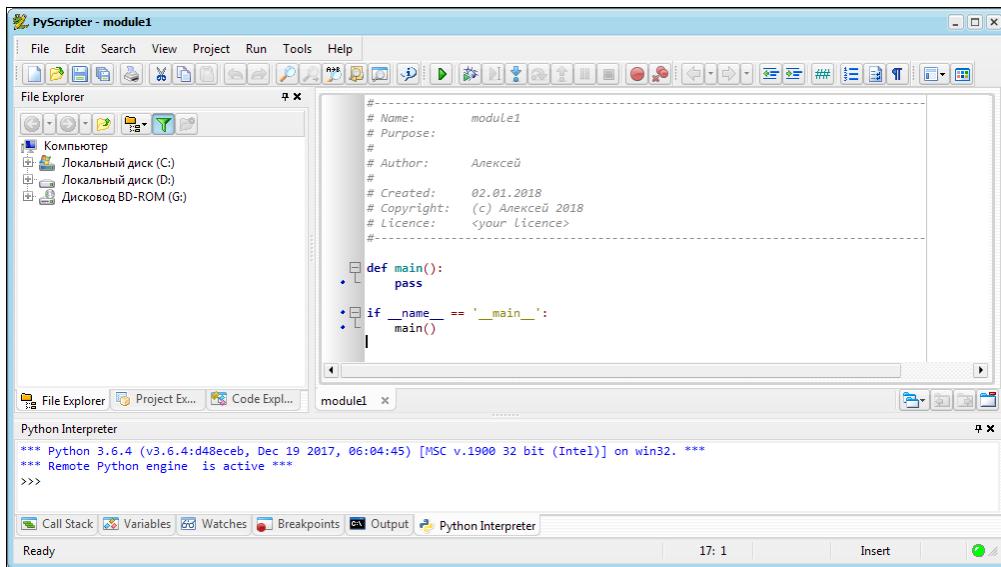
Робота з середовищем PyScripter

План, що й казати, був чудовий: простий і ясний, краще не придумати. Недолік у нього був тільки один: було зовсім невідомо, як його виконати.

Л. Керром. «Аліса в Країні Див»

Одразу застерігаємо, що повністю описувати програму PyScripter ми не будемо: по-перше, можливості такої немає, а, по-друге, необхідності, якщо чесно, також. Тому ми зупинимося лише на тих налаштуваннях і режимах, які критичні й будуть (або можуть бути) корисними читачеві в процесі роботи над матеріалом книги (маються на увазі, насамперед, програмні коди, які розглядаються в книзі).

Передусім, варто звернути увагу, що інтерфейс середовища розробки PyScripter підтримує різні мови. На рис. В.17 показано вікно програми PyScripter із англомовним інтерфейсом.



*Рис. В.17. Вікно програми PyScripter із англомовним інтерфейсом
і шаблонним кодом у вікні редактора кодів*



За замовчуванням під час запуску програми PyScripter у внутрішньому вікні редактора кодів для нового, автоматично створеного (але ще не збережено-го) файла пропонується шаблонний код, як це можна побачити на рис. В.17. Цей шаблонний код можна видалити й увести власний. Також користувач може змінити налаштування програми, в тому числі й уміст шаблонного коду.

Якщо ми хочемо використати іншу мову для інтерфейсу, то в меню *View* слід вибрати підменю *Language*, а в цьому підменю — команду з назвою мови, яку ми обираємо (наприклад, *Russian*), як показано на рис. В.18.

Pyhton

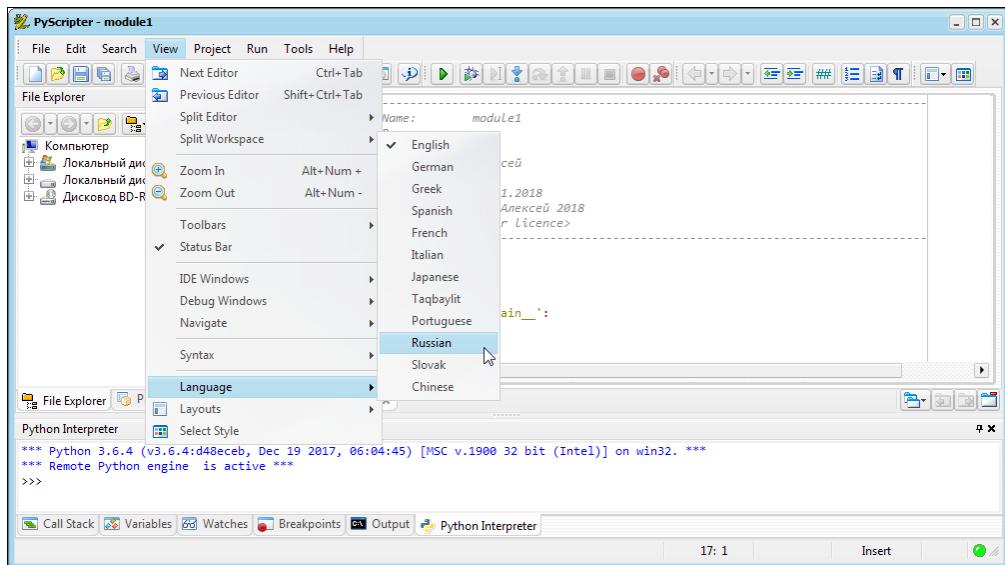


Рис. В.18. Вибір мови для інтерфейсу програми PyScripter

У результаті інтерфейс програми PyScripter стане (при виборі команди Russian) російськомовним. Щоб повернутися назад до англомовного інтерфейсу, в меню Вид у підменю Язык вибираємо команду Английский (рис. В.19).

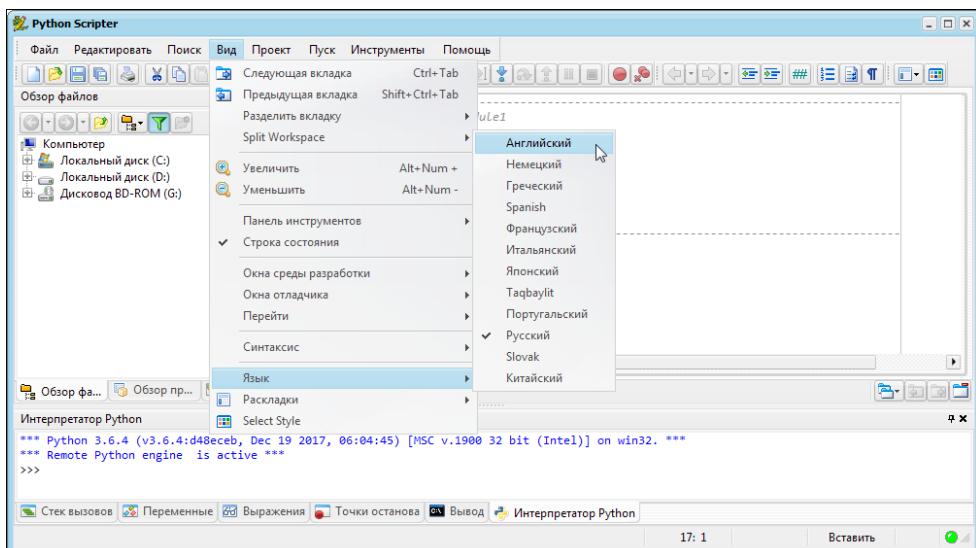


Рис. В.19. Вікно програми PyScripter із російськомовним інтерфейсом

Сподіваємося, що такі стандартні процедури, як створення, збереження, відкриття й закриття робочого документа (файла з програмою), у читача проблем не викличуть. Разом із тим звертаємо увагу на досить корисну утиліту: внутрішнє вікно File Explorer, безпосередньо в системі каталогів якого можна вибрати той документ, із яким зираємося працювати (рис. В.20).

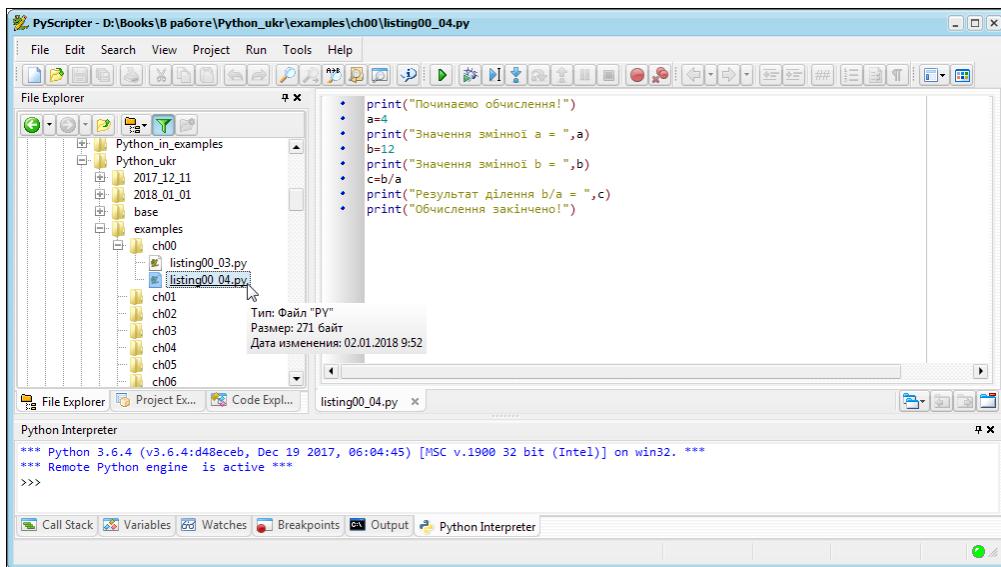


Рис. В.20. Вибір потрібного файла з програмним кодом безпосередньо у вікні програми PyScripter

Середовище розробки PyScripter доволі гнучке в плані налаштувань. Наприклад, щоб налаштувати параметри редактора кодів (такі, скажімо, як шрифт або режим виділення синтаксичних конструкцій) у меню Tools в підменю Options вибираємо команду Editor Options, як показано на рис. В.21.

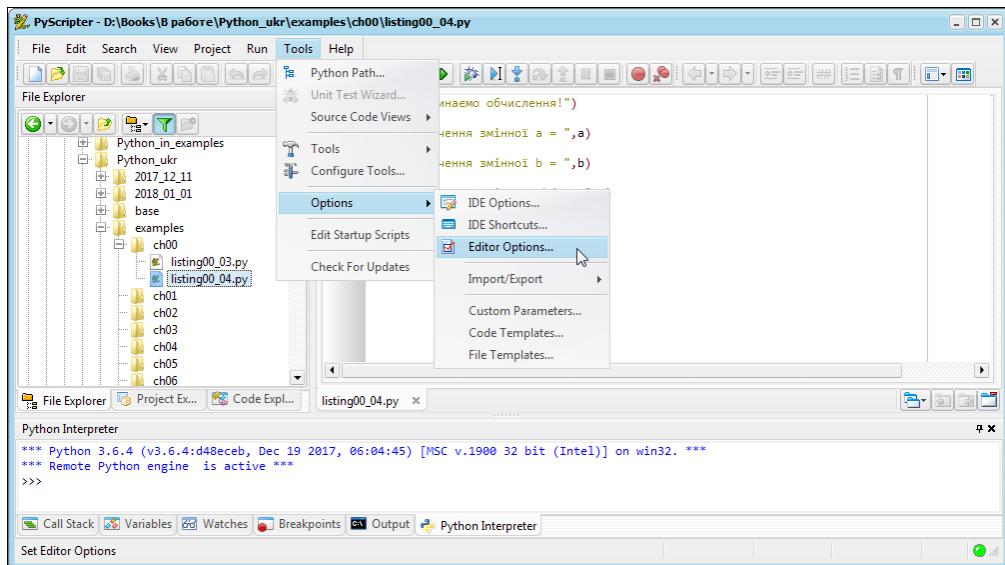


Рис. В.21. Перехід у режим налаштування параметрів редактора програмних кодів

У результаті відкриється діалогове вікно з назвою Editor Options, у якому, власне, і виконуються налаштування (рис. В.22).

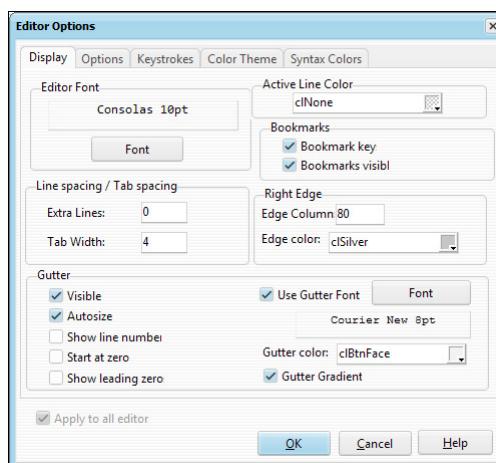


Рис. В.22. Вікно налаштування параметрів редактора програмних кодів

Одне досить важливе зауваження стосується програмних кодів, у яких використовується кириличний текст. Такі файли треба зберігати

у «правильному» кодуванні — інакше під час наступного відкриття в тому місці, де був кириличний текст, з'являться «карлючки». Зокрема, рекомендується перед збереженням файла в меню Edit у підменю File Format установити кодування UTF-8 (рис. B.23).

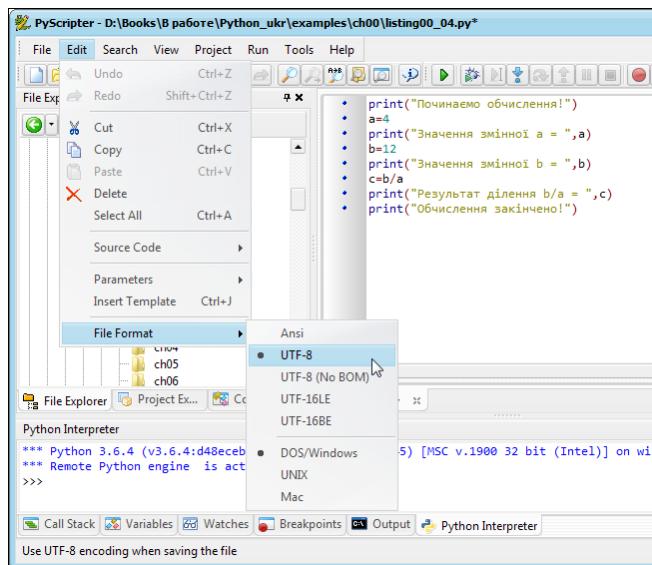


Рис. B.23. Збереження файлів із кириличним текстом у «правильному» кодуванні для подальшої коректної роботи

Також раніше ми відзначали, що використовувані за замовчуванням шаблони програмних кодів, що пропонуються користувачеві в різних ситуаціях (при створенні нового документа), можна змінити (відредактувати). Корисними в таких випадках будуть команди підменю Options в меню Tools. На рис. B.24 показано діалогове вікно File Templates, за допомогою якого редагуються шаблони.

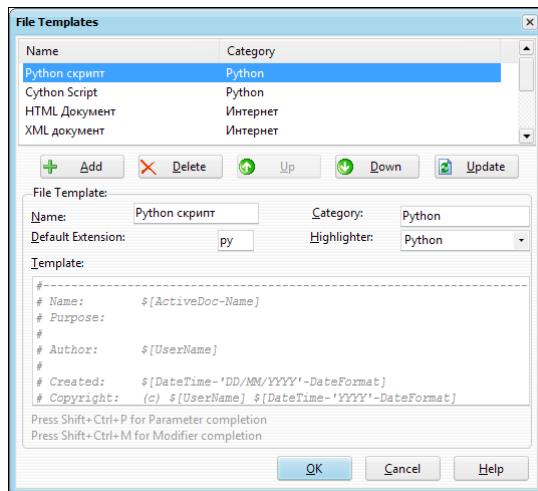


Рис. В.24. Вікно налаштування шаблонів

Характеризуючи ситуацію в цілому, варто сказати, що середовище PyScripter налаштовувати доволі просто, тому на практиці проблем із налаштуванням зазвичай не виникає навіть у непідготовлених користувачів. У разі нагальної потреби, зрозуміло, завжди можна звернутися до додівкової системи середовища розробки PyScripter.



У книзі ми будемо розглядати вихідні програмні коди на мові Python і результат виконання цих кодів – у текстовому форматі. Тому роботу із середовищем розробки окремо обговорювати не будемо. Фактично, задача набору виконання програмного коду повністю лягає на плечі читача. Добре, що завдання це не надто складне.

Якщо все ж таки виникнуть непередбачені проблеми з програмним захистом, нагадаємо, що в найгіршому разі програмний код можна набрати у звичайному текстовому редакторі й зберегти його у відповідний файл із розширенням `ру`. Потім цей файл виконується за допомогою програми-інтерпретатора. Тобто програму-інтерпретатор все ж таки доведеться встановити.

Подяка

Якби кожна людина займалася
своєю справою, Земля крутила-
ся б швидше.

Л. Керролл. «Аліса в Країні Див»

Автор висловлює щиру вдячність *Ілоні Васильєвій*, без напруженості і плідної праці та всебічної підтримки якої ця книга не вийшла б українською мовою.

Приклади й задачі з книг із програмування звичайно проходять апробацію на терплячій, але вимогливій аудиторії — у студентах і слухачах курсів. Їм — окрема подяка. Адже безпосереднє спілкування з тими, хто навчається програмуванню, дає безцінний досвід для викладача, і це, звісно, знаходить своє відображення в книгах.

Величезної подяки заслуговують читачі, які надсилали і надсилають листи зі своїми критичними зауваженнями й пропозиціями. Завдяки їхньому небайдужому ставленню втілено в життя багато чудових ідей. Від шановного читача цілком залежить, чи триватиме цей процес.

Зворотний зв'язок

Якщо в світі все безглаздо, – сказала Аліса, – що заважає вигадати якийсь сенс?

Л. Керролл. «Аліса в Країні Див»

Свої пропозиції і зауваження читачі можуть направляти електронною поштою alex@vasilev.kiev.ua або vasilev@univ.kiev.ua. Книги, в першу чергу, пишуться для читачів. Тому вкрай важливо знати, що в книзі вдалося, а що — ні. Конструктивна критика є зворотний зв'язок із читачами — дуже дієві інструменти, які дозволяють рухатися у правильному напрямку.

Деяку корисну інформацію щодо цієї та інших книг автора представлено на сайті www.vasilev.kiev.ua. Якщо ж потрібної читачам інформації там немає, але її розміщення технічно можливе, є сенс написати листа з пропозицією додати на сторінку відсутні відомості — адреси електронної пошти наведено вище.



Розділ 1

Програма мовою Python

.....

*Усе, що сказано три рази,
стає істиною.*

Л. Керролл. «Аліса в Країні Див»

Ми розпочинаємо вивчення мови Python. У цьому розділі познайомимося з базовими конструкціями, які дозволять нам створювати нескладні програмні коди. Деякі моменти спочатку пояснюватимуться на дещо поверхневому рівні, аби полегшити для новачків процес переходу від теоретизування до практичного використання Python для написання програм.



Нерідко програми, написані мовою Python, називають **сценаріями**. Ми трохи відійдемо від традиції й термін сценарій уживати не будемо.

Почнемо ми з того, що обговоримо загальні принципи організації програмного коду, написаного на мові Python.

Розмірковуючи про програму

- Скажіть, будь ласка, куди мені звідси йти?
- А куди ти хочеш потрапити? – відповів Кіт.
- Мені все одно – сказала Аліса.
- Тоді все одно, куди і йти – зауважив Кіт.

Л. Керролл. «Аліса в Країні Див»

Програма в Python — це послідовність команд. Жодних спеціальних інструкцій для формального позначення початку чи кінця коду програми використовувати не треба. Таким чином, під час написання програми ми розміщуємо команди одна за одною — у тому порядку, як вони повинні виконуватися. Команда розташовується на початку рядка. Відступів робити не потрібно. В кожному рядку зазвичай по одній команді — тобто кожна команда в новому рядку.



У цьому разі йдеться про «прості» команди, на зразок команди виводу в консольне вікно текстового повідомлення. Як зазначалося вище, ці команди розташовані на початку рядка, відступ (пробіл) перед ними не ставиться, і в кінці команди теж нічого ставити не треба. Трохи згодом ми познайомимося з інструкціями керування — такими, як оператор циклу й умовний оператор. У цих синтаксических конструкціях за допомогою пробілів виділяються складові частини виразу для оператора (блок команд тіла оператора). Але про це поговоримо пізніше. На даний момент важливо запам'ятати, що пробіл у мові Python — важливий елемент із точки зору синтаксису й використовувати його слід надзвичайно обережно. Зайвий пробіл може зумовлювати помилку.

У принципі, це все, що нам поки що необхідно знати про структуру програми для того, щоб розпочати написання програмного коду. З часом ми познайомимося з різними синтаксичними конструкціями, які вносять у програмний код велику інтригу й роблять процес програмування цікавим і часом непередбачуваним. Але це буде пізніше.

Що важливо зараз? Зараз важливо розуміти, що команди в програмі повинні бути коректними, як мінімум, з точки зору синтаксису мови Python. Проте навіть формально правильні команди не гарантують правильності виконання програми. Зазвичай програми пишуть для розв'язання тієї чи іншої задачі. Задача розв'язується відповідно до певного алгоритму, який розроблює й реалізує переважно програміст. Тобто в програмі реалізується деякий алгоритм. Якщо цей алгоритм неправильний, то і програма видасть не той результат, якого від неї очікують. Тому написання програми починається з розробки алгоритму, а вже потім цей алгоритм реалізується у вигляді команд програми. Ми для простоти входимо з припущення, що алгоритм є, він — правильний, і його лише необхідно «перекласти» на мову інструкцій Python.

Практично будь-яка програма оперує деякими даними. Це може бути як реально велика база даних, так і одне-едине значення — принципової різниці тут немає. Важливо те, що дані в програмі повинні якось зберігатися. Ми поки що будемо «зберігати» дані за допомогою *zmінних*. Щодо змінних є два важливих моменти:

- у змінної є ім'я (або назва);
- змінна *посилається* на деяке значення.

У принципі, значення, на яке посилається змінна, ми можемо:

- прочитати;
- змінити.

І в тому, і в іншому випадках ми використовуємо у контексті команди ім'я змінної. Оскільки в Python тип змінної явно не зазначають (він визначається за значенням, на яке посилається змінна), то попередньо оголошувати змінні не потрібно. Просто за першого використання змінної їй одразу присвоюється значення.



Зазвичай, коли пояснюють призначення змінних, порівнюють їх, наприклад, із кошком або банківською коміркою. Значення змінної за такої аналогії — це те, що знаходиться в кошику/комірці. Поки що ми можемо думати про змінну саме так. Проте у Python справи зі змінними виглядають трохи інакше. Технічно змінна містить адресу в області пам'яті, де зберігається значення, яке ототожнюється зі значенням змінної. Втім, дуже часто (у простих випадках) зовнішній ефект такий, неначе б змінна реально містила своє значення — як кошик чи банківська комірка. На даному етапі механізм збережання значень змінних не важливий. Трохи згодом, коли це питання стане актуальним, ми розставимо всі крапки над «і».

Спочатку ми розглянемо невелику програму. У ній, крім декількох простих команд, будуть використані коментарі. *Коментар* — це текст, призначений для програміста. Інтерпретатором коментар ігнорується. Для створення коментаря використовують символ #. Все, що праворуч від символу #, є коментарем.

Приклад простої програми

Треба бігти з усіх ніг, щоб тільки залишатися на місці. А щоб кудись потрапити, треба бігти як мінімум удвічі швидше!

Л. Керролл. «Аліса в Країні Див»

У програмі, яку ми розглянемо далі, реалізується такий алгоритм:

- виводиться текстове привітання з проханням до користувача вказати ім'я;
- після того, як користувач уводить ім'я, воно читається і записується у змінну;
- програма виводить ще одне повідомлення, а текст повідомлення містить ім'я, яке на попередньому кроці ввів користувач.

Для виведення повідомень у консольне вікно використовується функція `print()`, а для зчитування введеного користувачем тексту (ім'я користувача) використовуємо функцію `input()`. Також у програмі є коментарі. Повний код програми наведено в лістингу 1.1.



Лістинг 1.1. Програма з уведенням і виведенням даних

```
# Виводиться повідомлення
print("Давайте познайомимося!")
# Зчитуємо введене користувачем значення.
# Результат записується у змінну name
name=input("Як Вас звати? ")
# Виводиться нове повідомлення
print("Доброго дня, " + name + "!")
```

Розділ 1. Програма мовою Python

Оскільки це перша наша «офіційна» програма, з'ясуємо питання про те, як вона буде виконуватися в середовищі PyScripter та IDLE (на практиці між процесами виконання програми в цих середовищах є деякі відмінності). На рис. 1.1 показано вікно середовища PyScripter із програмним кодом (перед запуском програми на виконання).

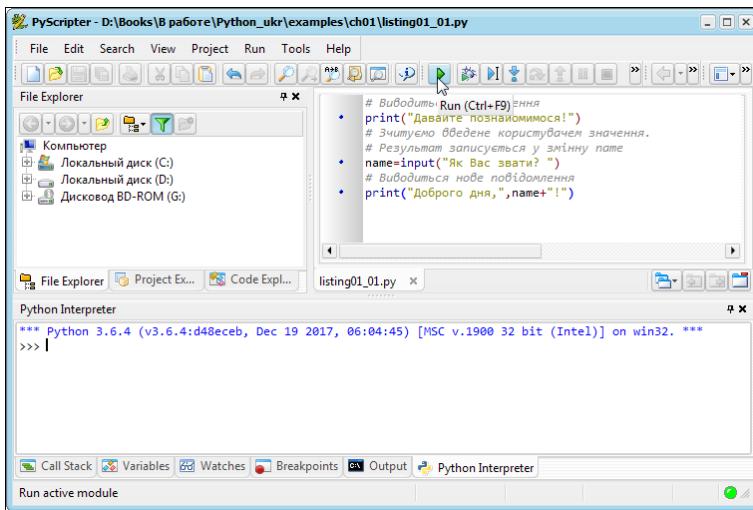


Рис. 1.1. Вікно середовища PyScripter перед початком виконання програми

У процесі виконання програмного коду спочатку у вікні інтерпретатора з'являється перше повідомлення, а потім відкривається вікно з полем уводу. Перед полем уводу відображується текст Як Вас звати?, а в полі вводу ми вказуємо ім'я користувача (рис. 1.2).

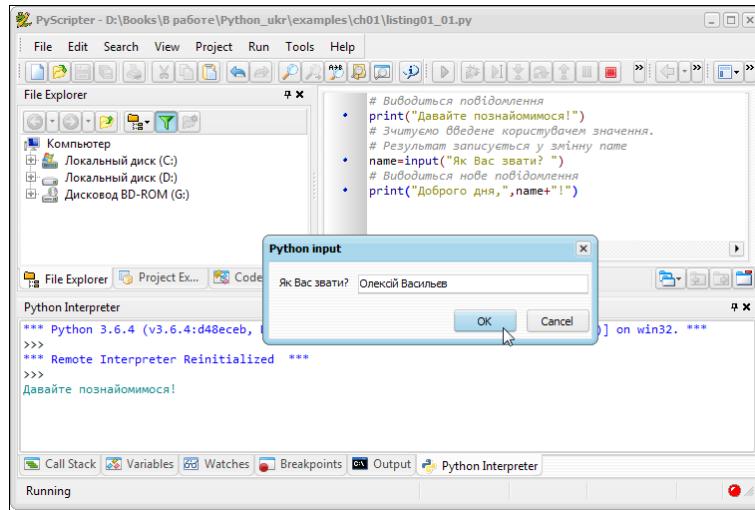


Рис. 1.2. Поява вікна з полем уводу в процесі виконання програми в середовищі PyScripter

Після натискання на кнопці ОК у вікні вводу отримуємо результат, як на рис. 1.3.

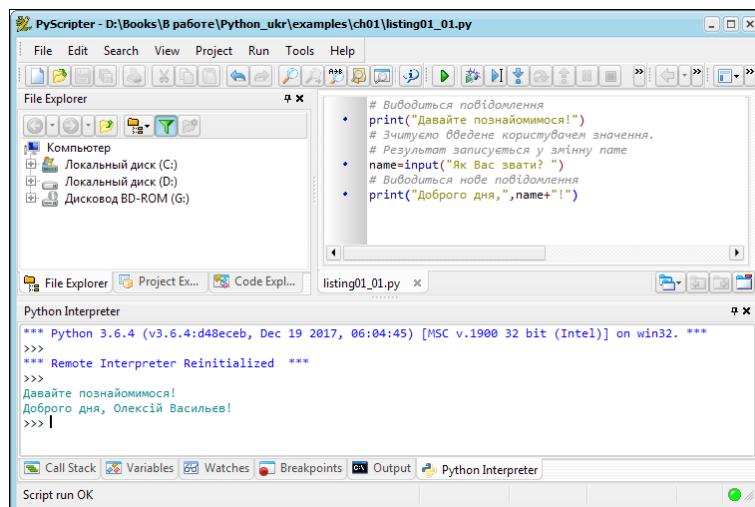


Рис. 1.3. Результат виконання програми в середовищі PyScripter

Таким чином, під час роботи із середовищем PyScripter уведення виконується в окремому вікні, яке відображується автоматично. Трохи інша

Розділ 1. Програма мовою Python

ситуація з використанням середовища IDLE. На рис. 1.4 показано вікно середовища з програмним кодом перед початком виконання програми.

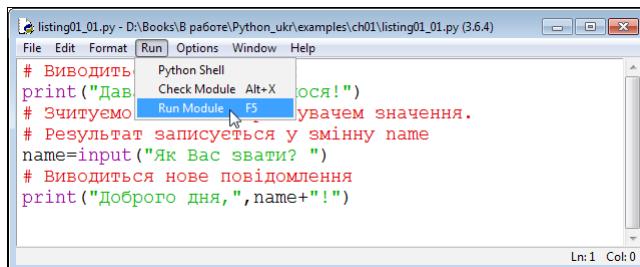


Рис. 1.4. Вікно середовища IDLE перед початком виконання програми

Після запуску програми на виконання у вікні інтерпретатора з'являється перше повідомлення і в новому рядку фраза Як Вас звати?, після якої користувач уводить текст, як це показано на рис. 1.5.

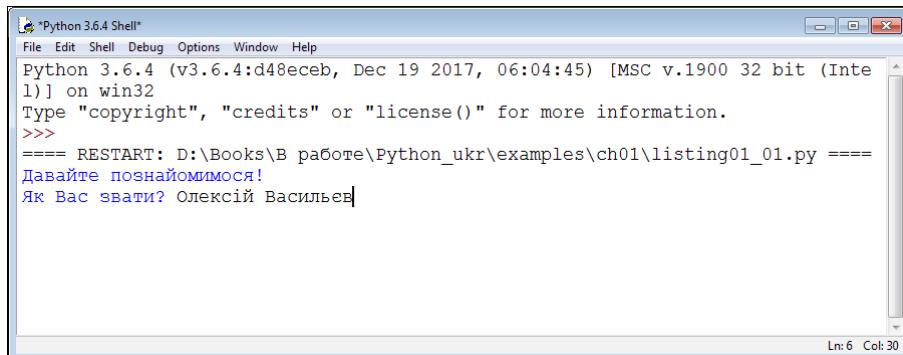


Рис. 1.5. У процесі виконання програми в середовищі IDLE введення тексту відбувається в консольному вікні

Уведення тексту підтверджується натисканням клавіші <Enter>. Результат виконання програми зображенено на рис. 1.6.

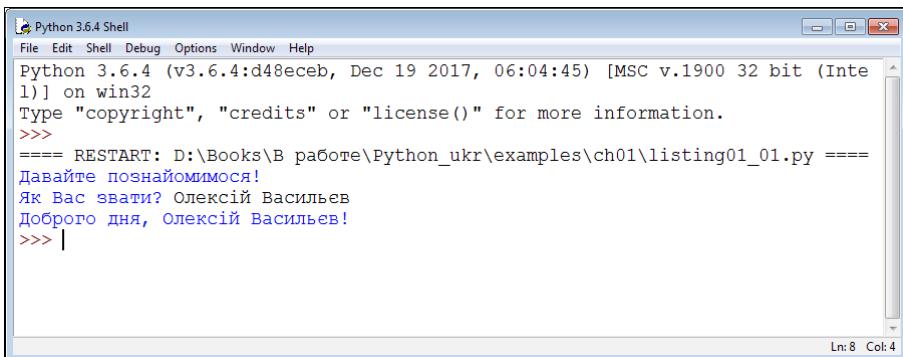


Рис. 1.6. Результат виконання програми в середовищі IDLE

Отже, результат виконання програми буде таким (жирним шрифтом виділений уведений користувачем текст):

■ Результат виконання програми (з лістингу 1.1)

Давайте познайомимося!
Як Вас звати? **Олексій Васильєв**
Доброго дня, Олексій Васильєв!



Якщо програма виконується в середовищі IDLE, результат буде достеменно таким, як показано вище (зрозуміло, з урахуванням того, яке значення ввів користувач). У середовищі PyScripter другий рядок відсутній. Хоча ми передбачаємо використовувати середовище PyScripter, результат для тих програм, у яких користувач уводить дані, будемо наводити у вигляді, як для середовища IDLE.

Тепер розглянемо детальніше програмний код, виконання якого приводить до такого результату (див. лістинг 1.1):

- Все, що починається із символу #, є коментарем і на результат виконання програми не впливає.
- Командою print("Давайте познайомимося!") у вікні інтерпретатора виводиться повідомлення, що визначається аргументом функції print().
- У команді name=input("Як Вас звати? ") викликається функція input() із текстовим аргументом, а результат виклику функції записується в змінну name. Як наслідок, у вікні

інтерпретатора (вікні з полем уводу) з'являється текст, переданий як аргумент функції `input()`. Текст, яке користувач увів у вікні інтерпретатора (поле виводу діалогового вікна) повертається як значення (результат) функції `input()`. Це і є значення змінної `name`.

- Командою `print("Доброго дня, " + name + " !")` у вікні інтерпретатора виводиться ще одне повідомлення, отримане об'єднанням (конкатенацією) тексту Доброго дня, значення змінної `name` і знака оклику.



В останній команді у функції `print()` два аргументи: "Доброго дня," і `name + " ! "`. І перший, і другий аргументи — текстові. Вони виводяться в одному рядку один за одним, причому між ними автоматично, за замовчуванням, додається пробіл. Щодо другого аргументу, то формально він представлений як «сума» текстової змінної `name` (точніше, змінної з текстовим значенням) і тексту " ! ". Якщо операція додавання застосовується до текстових значень, результатом є текст, отриманий об'єднанням відповідних текстів. Зверніть увагу, що пробіл у цьому випадку не додається. Щоб не запутатися з додаванням або не додаванням пробілу можна користуватися таким правилом: функція `print()` друкує свої аргументи через пробіл, а при об'єднанні рядків пробіл не додається. Ну ѹ, зрозуміло, читач уже помітив, що текстові значення вказують у подвійних лапках. Це не єдиний спосіб виділення тексту в Python (можна, наприклад, використовувати одинарні лапки). Ми, в основному, будемо брати текстові значення (літерали) у подвійні лапки.

Далі ми детальніше обговоримо деякі особливості роботи зі змінними в Python.

Обговорюємо змінні

Ніколи не вважай себе не таким,
яким тебе не вважають інші, і тоді
інші не визнають тебе не таким,
яким ти хотів би їм здаватися.

Л. Керролл. «Аліса в Країні Див»

Ми вже мали справу зі змінними. Але це було швидше побіжне знайомство. Тут ми приділимо змінним трохи більше уваги. Треба сказати, вони того варти. Тим більше, що в Python змінні досить специфічні.

Насамперед, щодо назви змінних: у принципі, це може бути практично будь-яке ім'я (комбінація букв, цифр і символів підкреслювання), яке не збігається з жодним із ключових слів Python. Список ключових слів Python наведено в таблиці 1.1.

Таблиця 1.1. Ключові слова Python

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Ключові слова не можуть бути модифіковані, і спроба використати змінну з відповідним іменем призведе до помилки.



Зауважте, що в цьому випадку ми не обговорюємо призначення ключових слів. Список ключових слів наведено виключно для того, щоб читач знов, як не варто називати змінні.

Окрім ключових слів, у Python існує багато будованих ідентифікаторів — таких, наприклад, як назви бібліотечних функцій (список будованих ідентифікаторів наведено в таблиці 1.2).

Таблиця 1.2. Будовані ідентифікатори Python

ArithmetricError	SyntaxError	float
AssertionError	SyntaxWarning	format
AttributeError	SystemError	frozenset
BaseException	SystemExit	getattr
BlockingIOError	TabError	globals
BrokenPipeError	TimeoutError	hasattr
BufferError	True	hash
BytesWarning	TypeError	help
ChildProcessError	UnboundLocalError	hex
ConnectionAbortedError	UnicodeDecodeError	id
ConnectionError	UnicodeEncodeError	input
ConnectionRefusedError	UnicodeError	int
ConnectionResetError	UnicodeTranslateError	isinstance
DeprecationWarning	UnicodeWarning	issubclass
EOFError	UserWarning	iter
Ellipsis	ValueError	len
EnvironmentError	Warning	license
Exception	WindowsError	list
False	ZeroDivisionError	locals
FileExistsError	__build_class__	map
FileNotFoundException	__debug__	max
FloatingPointError	__doc__	memoryview
FutureWarning	__import__	min
GeneratorExit	__loader__	next
IOError	__name__	object
ImportError	__package__	oct
ImportWarning	abs	open

IndentationError	all	ord
IndexError	any	pow
InterruptedError	ascii	print
IsADirectoryError	bin	property
KeyError	bool	quit
KeyboardInterrupt	bytearray	range
LookupError	bytes	repr
MemoryError	callable	reversed
NameError	chr	round
None	classmethod	set
NotADirectoryError	compile	setattr
NotImplemented	complex	slice
NotImplementedError	copyright	sorted
OSError	credits	staticmethod
OverflowError	delattr	str
PendingDeprecationWarning	dict	sum
PermissionError	dir	super
ProcessLookupError	divmod	tuple
ReferenceError	enumerate	type
ResourceWarning	eval	vars
RuntimeError	exec	zip
RuntimeWarning	exit	
StopIteration	filter	



У таблиці 1.2 наведено, в основному, назви вбудованих класів виключень і назви вбудованих функцій. Взагалі, дізнатися, які ідентифікатори на цей момент задіяні в роботі інтерпретатора (визначені в програмі), можна за допомогою функції `dir()`. Якщо викликати цю функцію без аргументів (скажімо, скористатися командою `print(dir())`), отримаємо список (набір) ідентифікаторів (імен), які вже «зайняті». Проте до цього списку не будуть входити назви вбудованих функцій та інших убудованих ідентифікаторів. Отримати доступ до списку вбудованих ідентифікаторів можна через модуль `builtins`. Для цього необхідно за допомогою команди `import builtins` підключити модуль, а потім викликати функцію `dir()` з аргументом `builtins` (для відображення списку вбудованих ідентифікаторів в області виводу використовуємо команду `print(dir(builtins))`). Докладніше про підключення модулів розповідається в кінці цього розділу.

Формально ми можемо задіяти змінну з іменем, що збігається з тим чи іншим ідентифікатором. Однак, це може несподіваним чином вплинути на виконання програми. Подібні ситуації вважають поганим тоном у програмуванні, і їх слід уникати.

Ім'я змінної не може починатися з цифри. Також дуже обережно варто використовувати підкреслювання на початку і в кінці імені змінної (змінні з початковим підкреслюванням і з подвійним підкреслюванням на початку і в кінці імені обробляються за спеціальними правилами).

Про те, що для змінних не потрібно явно оголошувати тип, ми вже знаємо. Однак, це зовсім не означає, що типів даних в Python немає. Інша справа, що таке поняття як *тип* ототожнюється саме з даними, а не зі змінною.



Узагалі тип даних важливий, принаймні, з двох причин: тип даних визначає обсяг пам'яті, що виділяється для зберігання цих даних, а також перебуває у тісному зв'язку з тими операціями, які допустимі з даними. У багатьох мовах програмування змінні оголошуються із зазначенням типу. Ось у цьому випадку змінну зручно уявляти як кошик, на якому написана назва змінної, а значення змінної — те, що всередині кошика. Таких кошиків у програмі може бути багато. Кожний кошик ототожнюється з якоюсь конкретною змінною, у кожному кошику щось «лежить» (значення відповідної змінної). Розмір кошика визначається типом змінної, з якою ми цей кошик ототожнюємо. Коли ми читуємо значення змінної, то просто «дивимося», що лежить у кошику. Коли змінюємо значення змінної, то виймаємо з кошика те, що в ньому було, і поміщаємо туди новий уміст.

У Python все трохи інакше. Змінні не мають типу. Але тип є у даних, які «запам'ятовуються» за допомогою змінних. Тому концепція звичайних кошиків втрачає свою актуальність. Актуальною стає концепція кошиків, до яких прив'язані мотузочки. Змінна — це мотузочка з биркою (назва змінної), а кошик — це дані, на які посилається змінна. Тобто тепер назва змінної — це не бирка на кошику, а бирка на мотузочці. На кошиках бирок немає. Як і раніше, розмір кошика визначається характером його вмісту (тип даних). Але доступ до кошика у нас є тільки через мотузочку. Мотузочки всі однакові й відрізняються лише назвою на бирці. Що ми можемо зробити в такій ситуації? Ми можемо потягнути за мотузочку і подивитися, що міститься в кошику, до якого прив'язано мотузочку. Це аналог зчитування значення змінної. Також ми можемо поміняти вміст кошика чи взагалі відв'язати кошик від мотузочки і прив'язати мотузочку до іншого кошика. Це аналог зміни значення

змінної. Більше того, ми можемо прив'язати декілька мотузочек до одного його самого кошика. Мовою програмування це означає, що декілька змінних посилаються на одне й те саме значення. І таке в Python теж можливе.

У Python змінні *посилаються* на дані, а не містять їх, як у деяких інших мовах програмування. Кожна змінна «пам'ятає», у якому місці в пам'яті знаходиться деяке значення. Це значення ми ототожнюємо зі значенням змінної (хоча насправді значенням змінної є адреса комірки пам'яті з відповідними даними). А втім, коли ми звертаємося до змінної, то виконується автоматичний переход за посиланням на дані. Виникає ілюзія, як наче б змінна реально містила певне значення. Які наслідки усього сказаного? По-перше, описаний вище механізм нерідко є наріжним камнем у розумінні того, що відбувається під час виконання програмного коду і, зокрема, під час присвоювання значень змінним. По-друге, цілком очевидно, що одна й та сама змінна на різних етапах виконання програми може посыпатися не просто на різні значення, а й на значення різних типів (наприклад, спочатку посылатися на текст, а потім на число). По-третє, хоча безпосередньо в змінних типу немає, ми можемо отримати доступ через змінну до значення, на яке вона посилається, і дізнатися його тип. Залишилося лише з'ясувати, які типи даних узагалі можливі в Python.



Для визначення типу значення, на яке посилається змінна, можна використовувати функцію `type()`. Змінна вказується як аргумент функції.

Наш досвід роботи зі змінними поки що обмежується текстовими й числовими значеннями (приклад у *Вступ*). Так ось, текстові значення належать до типу, який називається `str`. Числові значення — до типу `int` (якщо це цілі числа) або типу `float` (якщо це дійсні числа у форматі значення з плаваючою крапкою). Приємною несподіванкою для любителів математичних розрахунків буде те, що в Python є вбудована підтримка для комплексних чисел. Для цих цілей використовують значення типу `complex`. Деякі приклади нескладних числових розрахунків наведено далі в цьому розділі.

Існує всього два значення логічного типу (*істина* або *хиба*). У Python логічний тип позначається як `bool`. Із логічними виразами ми познайомимося, коли почнемо вивчати умовні інструкції та інструкції циклу.

Трохи пізніше ми розглянемо *списки*, які відіграють роль масивів у Python. Списки належать до типу `list`. Ще в Python є *множини* (про них ми теж поговоримо, але не в цьому розділі), що належать до типу `set`. Існують й інші типи даних, які ми будемо вивчати поступово, у міру нашого опанування мовою Python.



Насамперед, маються на увазі такі «різновиди» даних, як кортежі (тип `tuple`) і словники (тип `dict`). А ще в Python є такі типи: `bytes` (незмінювана послідовність байтів), `bytearray` (змінювана послідовність байтів), `frozenset` (незмінювана множина), `function` (функція), `module` (модуль), `type` (клас). Є також типи для спеціальних значень `ellipsis` (для елемента `Ellipsis`) і `NoneType` (для значення `None`).

Як зазначалося вище, від типу даних залежить те, які операції можуть із цими даними виконуватися. Маніпулювати даними можемо або за допомогою функцій, або за допомогою операторів. Які оператори, коли і як використовуються в Python, ми обговоримо в наступному пункті.



На завершення цього пункту відзначимо, що в Python змінні можна не тільки створювати, а й видаляти. Для видалення змінної використовується інструкція `del`, після якої через пробіл зазначається ім'я змінної, яка видаляється. Якщо змінних, що видаляються, декілька, їх розділяють комою.

Основні оператори

Якби це було так, це б ще нічого.
Якби, звичайно, воно так і було.
Але так як це не так, так воно
і не так. Така логіка речей.

Л. Керрол. «Аліса в Країні Див»

Зазвичай виділяють чотири групи операторів:

- арифметичні;
- побітові;
- логічні оператори;
- оператори порівняння.

Арифметичні оператори призначені, насамперед, для виконання арифметичних розрахунків. У таблиці 1.3 перераховано й коротко описано основні арифметичні оператори мови Python. Одразу слід зауважити, що дія арифметичних операторів досить точно відповідає «математичній природі» цих операторів. При цьому ми припускаємо, що йдеться про числові розрахунки.

Таблиця 1.3. Арифметичні оператори

Оператор	Опис
+	Оператор додавання. Розраховується сума двох чисел
-	Оператор віднімання. Розраховується різниця двох чисел
*	Оператор множення. Розраховується добуток двох чисел
/	Оператор ділення. Розраховується відношення двох чисел
//	Оператор цілочислового ділення. Розраховується ціла частина від ділення одного числа на інше

Оператор	Опис
%	Оператор обчислення остачі від ціличислового ділення. Розраховується остача від ділення одного числа на інше
**	Оператор піднесення до степеня. Результатом є число, отримане піднесенням першого операнда до степеня, що визначається другим операндом



Деякі з перерахованих вище операторів можуть застосовуватися не тільки до значень числових типів, а й, наприклад, до тексту чи до списків. Цих тем ми поки не торкаємося й поговоримо про них пізніше. Нагадаємо тільки, що якщо до одного текстового рядка додати інший текстовий рядок (за допомогою оператора +), то в підсумку відбудеться конкатенація (об'єднання) рядків: результатом буде текст, отриманий унаслідок об'єднання рядків, що додаються. Наприклад, у результаті виконання команди `txt="Мова "+"Python"` змінна `txt` буде посыпатися на текстове значення "Мова Python".

Якщо ми спробуємо помножити текстове значення на ціле число (або ціле число на текст), то матимемо текстовий рядок, отриманий повторенням (ї конкатенацією) вихідного рядка (кількість повторів визначається ціличисловим операндом в інструкції множення тексту й числа). Наприклад, у результаті виконання команди `txt="Python "*3` змінна `txt` посылатиметься на текст "Python Python Python ", отриманий трикратним повторенням і конкатенацією вихідного тексту "Python ".

Приклад програмного коду для виконання нескладних арифметичних розрахунків наведено в лістингу 1.2.

Лістинг 1.2. Арифметичні оператори

```
a=(5+2)**2-3*2 # Результат 43
b=6-5/2          # Результат 3.5
c=10//4+10%3    # Результат 3
# Результати обчислень виводимо на екран
print("Результати обчислень:")
print(a,b,c)
```

Результат виконання цього програмного коду подано нижче:

■ Результат виконання програми (з лістингу 1.2)

Результати обчислень:

43 3.5 3



Можливо, деяких пояснень потребує процедура обчислення значення виразу $10//4+10\%3$. Так, значенням виразу $10//4$ є ціла частина від ділення 10 на 4 – це число 2. Значення виразу $10\%3$ – це число 1 (остача від ділення 10 на 3). У результаті отримуємо суму 2 і 1 – тобто число 3.

Мова Python дозволяє створювати дуже елегантні програмні коди. Тут ми скористаємося можливістю, щоб проілюструвати це твердження. Розглянутий вище програмний код ми перепишемо трохи інакше. Зокрема, застосуємо функцію `eval()`, яка дозволяє обчислювати вирази, задані в текстовому форматі. Наприклад, якщо певний алгебраїчний вираз, записаний відповідно до правил синтаксису мови Python, взяти у подвійні лапки, то вийде текст. Якщо цей текст тепер передати аргументом функції `print()`, то на екрані з'явиться відповідний алгебраїчний вираз. Якщо ж текст (із алгебраїчним виразом) передати аргументом функції `eval()`, то відповідний алгебраїчний вираз буде обчислено. Звернімося до лістингу 1.3.

■ Лістинг 1.3. Використання функції eval()

```
a = "(5+2)**2-3*2" # Текстове значення
b = "6-5/2"           # Текстове значення
c = "10//4+10%3"     # Текстове значення
# Результати обчислень виводимо на екран.
# Для "обчислення" текстових виразів
# використовуємо функцію eval()
print("Результати обчислень:")
print(a + " = ", eval(a))
print(b + " = ", eval(b))
print(c + " = ", eval(c))
```

У результаті виконання цього програмного коду отримуємо таке:

■ Результат виконання програми (з листингу 1.3)

Результати обчислень:

$(5+2)^{**2}-3^*2 = 43$

$6-5/2 = 3.5$

$10//4+10\%3 = 3$



Результати обчислень пояснимо на прикладі маніпуляцій зі змінною `a`, якій як значення присвоюється вираз `"(5+2)**2-3*2"`. Це текст. Якщо б ми не використовували подвійні лапки, був би звичайний арифметичний вираз. Подвійні лапки арифметичний вираз перетворюють у текст. Тому змінна `a` посилається на текстове значення. Якщо ми передаємо змінну `a` як аргумент функції `print()`, то на екрані з'явиться текстовий вміст, на який посилається змінна — як і повинно бути. Якщо ми вказуємо змінну `a` аргументом функції `eval()`, то здійснюється спроба обчислити вираз, представлений текстом, на який посилається змінна `a`. Функція `eval()` досить «розумна». Якщо її аргумент — звичайний текст (набір слів), то результатом буде цей самий текст. Якщо, як у нашому випадку, у тексті «заховано» арифметичний вираз — буде обчислено значення цього виразу. Взагалі, щоб зрозуміти, який буде результат виконання функції `eval()`, треба уявити, що виконується команда, представлена текстом в аргументі функції. Якщо задуматися, то легко зрозуміти, що ця проста обставина відкриває перед нами грандіозні перспективи. Наприклад, якщо ми послідовно виконаємо команди `x=3, y=7` і `z="x+y"`, а потім команду `print(z+" =", eval(z))`, то отримаємо в області виводу повідомлення `x+y = 10`. Чому так? Увесь секрет, очевидно, полягає в тому, як оброблюється значення змінної з функціями `print()` і `eval()`. У першому випадку нічого особливого не відбувається — текст `"x+y"` оброблюється як текст. В іншому випадку результатом буде значення виразу `x+y`, «схованого» в подвійних лапках. Оскільки до цього змінні `x` і `y` отримали числові значення, отримуємо суму двох числових значень.

Побітові (або двійкові) оператори дуже близькі до арифметичних у тому, що теж призначенні для роботи з числовими значеннями. Тільки у випадку побітових операторів обчислення виконуються на рівні двійкового коду числа. Для ефективного використання побітових операторів необхідно непогано розбиратися у двійковому зображеннях чисел. Наведені далі дані призначені для тих читачів, хто з цією темою знайомий не дуже добре.



У двійковому коді число зображене у вигляді послідовності нулів й одиниць. Старший біт використовується для визначення знака числа: нуль відповідає додатному числу, а одиниця – від'ємному числу.

У звичайному житті для запису чисел ми використовуємо десять цифр: 0, 1, 2 і так далі до 9 включно. За допомогою цих десяти цифр ми можемо записати будь-яке число. Досягається це завдяки позиційному зображеню чисел: число записується у вигляді послідовності цифр. Важливо, на якій позиції яка цифра знаходитьться. Припустімо, позиційне зображення числа має вигляд $\underline{a_n a_{n-1} \dots a_1 a_0}$ (рисочка зверху означає, що йдеться про позиційне зображення числа), де числові параметри a_0, a_1, \dots, a_n можуть набувати значення від 0 до 9 включно. Як визначити значення такого числа? Досить просто. За визначенням, має місце співвідношення $\underline{a_n a_{n-1} \dots a_1 a_0} = a_0 \cdot 10^0 + a_1 \cdot 10^1 + a_2 \cdot 10^2 + \dots + a_n \cdot 10^n$.

Таку саму схему ми можемо використовувати для того, щоб записувати числа за допомогою всього двох цифр – 0 і 1. Таке зображення називається двійковим кодом. Уявімо, що деяке число в позиційному зображенні має вигляд $\underline{b_n b_{n-1} \dots b_1 b_0}$, але тепер параметри b_0, b_1, \dots, b_n можуть набувати лише двох значень: 0 або 1. Чому дорівнює число $\underline{b_n b_{n-1} \dots b_1 b_0}$? Відповідь така: $\underline{b_n b_{n-1} \dots b_1 b_0} = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \dots + b_n \cdot 2^n$.

Основні арифметичні операції на рівні двійкового зображення чисел виконуються так само легко, як у звичному нам десятковому зображення. Тільки трохи видозмінюються основні правила. Наприклад, мають місце такі співвідношення: $0 + 0 = 0$, $1 + 0 = 1$, $0 + 1 = 1$, $1 + 1 = 10$ і так далі. Множення числа на 2 зводиться до додавання в позиційному зображенні цього числа в кінці нуля. Ділення на 2 парного числа означає «закреслювання» нуля в правій крайній позиції у двійковому коді (непарні числа на 2 без остачі не діляться).

Таким чином, будь-яке число ми можемо записати як послідовність нулів й одиниць і виконувати з цими числами всі ті операції, які ми звичайно виконували з числами в десятковому форматі. Але одне запитання залишається відкритим: як записувати від'ємні числа? Знову ж, якщо ми пишемо число на папері, ми просто додиємо перед ним знак «мінус». У принципі, це ж ми можемо зробити й у випадку, якщо число зображене двійковим кодом. Але проблема в тому, що аркуш паперу – не комп’ютер. Для комп’ютера потрібен інший підхід. Зазвичай застосовують принцип кодування від'ємних чисел, який називається «доповнення до нуля». Щоб зрозуміти його основну ідею, розгляньмо допоміжний приклад.

Розділ 1. Програма мовою Python

Припустімо, є якесь додатне (це важливо!) число, яке позначимо через x (ікс). А що таке число $-x$ (мінус ікс)? Відповідь може бути такою: це таке число, яке, додане до вихідного, дасть у результаті нуль. Тобто фактично, за визначенням, повинно бути так: $x + (-x) = 0$. Від цього й будемо відштовхуватися.

Є така операція, як побітова інверсія (у Python цій меті служить оператор `~`). Побітова інверсія полягає в тому, що у двійковому зображенні числа всі нули замінюються на одиниці, а всі одиниці замінюються на нулі. Якщо є деяке додатне число x , то число $\sim x$ отримаємо з числа x заміною нулів й одиниць на свої антиподи. Запитаймо себе: а що буде, якщо ми обчислим суму чисел x і $\sim x$? Неважко злагнути, що в результаті отримаємо число, двійкове зображення якого складається з одиниць. Справді, на тій позиції, де у числа x міститься 0, у числа $\sim x$ міститься 1. Там, де у числа x міститься 1, у числа $\sim x$ міститься 0. А ми вже знаємо, що $0 + 1 = 1 + 0 = 1$. Тобто отримаємо таке: $x + (\sim x) = \underbrace{111\dots111}_{n \text{ одиниць}}$. А тепер до отриманого результату додамо число 1 («у

гру вступає» правило $1 + 1 = 10$). Отримуємо таке: $x + (\sim x) = \underbrace{111\dots111}_{n \text{ одиниць}} + 1 = \underbrace{1000\dots00}_{n \text{ нуляв}}$. І ось тепер ми згадуємо, що йдеться не просто

про обчислення, а про обчислення на комп'ютері. А в комп'ютері для запам'ятовування чисел (у двійковому коді) виділяється фіксована кількість бітів — тобто фіксована кількість позицій (або розрядів) для запису числа. Якщо для запису числа потрібно більше розрядів, ніж це відведено в комп'ютері, старші розряди будуть загублені — вони просто ігноруються. Тепер уявімо, що для запису чисел використовується n розрядів. Тоді ми зможемо записати число x , число $\sim x$, не буде проблем із їхньою сумаю $x + (\sim x)$, але ось при записі результату $x + (\sim x) + 1$ уже знадобиться на один біт (розряд) більше. Але для цього біта місце не виділено. Тому в числі $x + (\sim x) + 1 = \underbrace{1000\dots00}_{n \text{ нуляв}}$ одиницю в старшому біті буде загублено, і ми отримаємо код

із n нуляв. Але це число 0! Таким чином, із урахуванням практики комп'ютерних обчислень, отримуємо результат $x + (\sim x) + 1 = 0$. Із іншого боку, ми раніше домовилися, що $x + (\sim x) = 0$. Який із цього висновок? Дуже простий: $-x = \sim x + 1$. Іншими словами, щоб отримати двійковий код від'ємного числа, потрібно взяти двійковий код відповідного додатного числа, по бітах інвертувати його і до отриманого значення додати одиницю. При цьому якщо у додатних чисел старший біт завжди нульовий, то у від'ємних чисел він завжди одиничний. Саме старший біт є ознакою додатності/від'ємності числа. Додатні числа з двійкового коду в десятковий переводяться за формулою $b_n b_{n-1} \dots b_1 b_0 = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \dots + b_n \cdot 2^n$. Якщо число від'ємне, то цю формулу застосовувати не можна. Щоб перевести від'ємне число у двійковий код у звичне нам десяткове зображення, необхідно виконати такі дії:

- по бітах інвертувати двійкове зображення від'ємного числа;

- додати до отриманого результату одиницю;
- отриманий двійковий код перевести в десяткове зображення за формуллю $b_n b_{n-1} \dots b_1 b_0 = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \dots + b_n \cdot 2^n$;
- дописати знак «мінус».

Щоб зрозуміти, звідки взялися ці правила, досить помітити, що, враховуючи особливості зображення чисел у комп’ютері, має місце співвідношення $-x + \sim(-x) + 1 = 0$. Наступні раздуми (з точністю до позначень) повторюють ті, що наводилися раніше.

Наприклад, ми хочемо дізнатися, яке двійкове зображення для числа -5 . Почнімо з того, що визначимо код для числа 5 . Нескладно перевірити, що це така послідовність бітів: $00\dots000101$. Після побітового інвертування отримаємо код $11\dots111010$. До цього коду додамо 1 , отримаємо код $11\dots111011$. Це і є двійковий код числа -5 .

Тепер навпаки, нам хочеться дізнатися, яке число закодовано у вигляді $11\dots110111$. Оскільки старший (найбільш крайній ліворуч) біт дорівнює 1 , число від’ємне. Щоб дізнатися, що ж це за число, виконаймо побітове інвертування коду. Отримаємо такий результат: $00\dots001000$. Додамо одиницю, отримаємо код $00\dots001001$. Це код числа 9 . Отже, у вихідній послідовності бітів $11\dots110111$ було закодовано число -9 .

Побітові оператори перераховано й описано у таблиці 1.4.

Таблиця 1.4. Побітові оператори

Оператор	Опис
\sim	<i>Побітова інверсія</i> (унарний оператор — у нього один операнд). Результатом є число, отримане заміною нулів на одиниці й одиниць на нулі в побітовому зображенні операнда (сам операнд при цьому не змінюється)
$\&$	<i>Побітове і</i> . При обчисленні результату порівнюються побітові зображення операндів. Якщо на одній і тій самій позиції в операндах стоять одиниці, то в числі-результаті на цій самій позиції буде одиниця. В іншому випадку (тобто якщо хоча б в одній із двох позицій нуль), у числі-результаті на відповідній позиції буде нуль

Оператор	Опис
	<i>Побітове або.</i> Порівнюються побітові зображення операндів. Якщо на одній і тій самій позиції в операндах стоять нулі, то в числі-результаті на цій самій позиції буде нуль. В іншому випадку (тобто якщо хоча б в одній із двох позицій одиниця), у числі-результаті на відповідній позиції буде одиниця
^	<i>Побітове виключне або.</i> Результат обчислюється шляхом порівняння побітових зображень операндів. Якщо на одній і тій самій позиції в операндах стоять різні значення (в одного числа нуль, а в іншого — одиниця), то в числі-результаті на цій позиції буде одиниця. В іншому випадку (тобто якщо на відповідних позиціях в операндах стоять однакові числа), у числі-результаті на цій позиції буде нуль
<<	<i>Зсув ліворуч.</i> Результат обчислюється так: у побітовому зображенні першого операнда виконується зсув ліворуч. Кількість розрядів, на які відбувається зсув, визначається другим операндом. Молодші відсутні біти заповнюються нулями
>>	<i>Зсув праворуч.</i> Для обчислення результату в побітовому зображенні першого операнда виконується зсув праворуч. Кількість розрядів, на які виконується зсув, визначається другим операндом. Біти ліворуч заповнюються значенням найстаршого біта (для додатних чисел це нуль, а для від'ємних — одиниця)

Щоб проілюструвати методи використання побітових операторів, розглянемо невеликий програмний код, поданий у лістингу 1.4.

Лістинг 1.4. Побітові оператори

```
a=70>>3
b=~a
c=a<<1
print(a,b,c)
print(7|3,7&3,7^3)
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 1.4)

```
8 -9 16
7 3 4
```



Про всяк випадок пояснимо результати виконання програмного коду. Результат виразу $70 >> 3$ – це число, отримане зсувом бітового зображення числа 70 на три позиції праворуч (із втратою молодших бітів). Двійковий код для числа 70 має вигляд 00...0001000**110** (три біти, які «зникають» після зсува праворуч, виділено жирним шрифтом). Після зсува на три позиції праворуч отримуємо 00...0000001000. Це код числа 8. Такий самий результат можна було отримати простіше, якщо згадати, що зсув праворуч на одну позицію еквівалентний ціличисловому діленню (діленню без остачі) на число 2. Якщо 70 тричі поділити без остачі на 2, отримаємо 8 (35 після першого ділення, 17 після другого ділення і 8 після третього ділення).

Далі, якщо застосувати побітове інвертування до числа 00...0000001000 (значення змінної a), отримаємо бінарний код 11...1111110111, який відповідає від'ємному числу -9. Хто бажає, може виконати перевірку самостійно, однак, якщо згадати, що результатом операції $\sim a + 1$ є код для значення $-a$ (у цьому випадку це -8), то нескладно здогадатися, що $\sim a$ відповідає значенню $-a - 1$ (тобто в цьому разі це -9).

Значення виразу $a << 1$ отримуємо зсувом бінарного коду для значення змінної a на одну позицію праворуч із заповненням молодшого біта нулем (відповідає множенню значення змінної a на 2). Оскільки значення змінної a дорівнює 8, то значення виразу $a << 1$ дорівнює 16.

У наступних виразах використовуються числа 7 (бінарний код 00...000111) і 3 (бінарний код 00..000011). Для побітових операцій | (або), & (i), ^ (виключне або) отримаємо такі результати:

$$\begin{array}{r} | \quad 00\dots000111 \\ | \quad 00\dots000011 \\ \hline 7 \quad 00\dots000111 \end{array}$$

$$\begin{array}{r} \& \quad 00\dots000111 \\ \& \quad 00\dots000011 \\ \hline 3 \quad 00\dots000011 \end{array}$$

$$\begin{array}{r}
 ^\wedge 00\dots000111 \\
 00\dots000011 \\
 \hline
 4 \quad 00\dots000100
 \end{array}$$

У лівому нижньому куті жирним шрифтом виділено результат відповідної операції в десятковій системі числення.

Із логічними значеннями ми зустрінемося під час перевірки умов в умовній інструкції (операторі). Значені у логічного типу всього два: `True` (*истина*) і `False` (*хиба*). Для роботи зі значеннями логічного типу призначенні спеціальні оператори, які прийнято називати логічними. Використовувані в Python *логічні оператори* описано в таблиці 1.5.

Таблиця 1.5. Логічні оператори

Оператор	Опис
<code>or</code>	Бінарний оператор (в оператора — два операнди) логічне або . У загальному випадку результатом виразу <code>x or y</code> є <code>True</code> , якщо значення хоча б одного з операндів <code>x</code> або <code>y</code> дорівнює <code>True</code> . Якщо значення обох операндів <code>x</code> і <code>y</code> дорівнюють <code>False</code> , результатом виразу <code>x or y</code> буде <code>False</code> . У Python вирази на основі оператора <code>or</code> обчислюються за спрощеною схемою: якщо перший операнд <code>x</code> інтерпретується як <code>True</code> , то <code>x</code> повертається як результат (другий операнд <code>y</code> при цьому не обчислюється). Якщо перший операнд <code>x</code> інтерпретується як <code>False</code> , то як результат повертається другий операнд <code>y</code> .
<code>and</code>	Бінарний оператор (в оператора — два операнди) логічне і . У загальному випадку результатом виразу <code>x and y</code> є значення <code>True</code> , якщо значення обох операндів <code>x</code> і <code>y</code> у дорівнюють <code>True</code> . Якщо значення хоча б одного з операндів <code>x</code> або <code>y</code> дорівнює <code>False</code> , результатом виразу <code>x and y</code> буде <code>False</code> . У Python вирази на основі оператора <code>and</code> обчислюються за спрощеною схемою: якщо перший операнд <code>x</code> інтерпретується як <code>False</code> , то <code>x</code> повертається як результат (другий операнд <code>y</code> при цьому не обчислюється). Якщо перший операнд <code>x</code> інтерпретується як <code>True</code> , то як результат повертається другий операнд <code>y</code> .

Оператор	Опис
not	<i>Логічне заперечення.</i> Унарний оператор (в оператора — один операнд). Результатом виразу <code>not x</code> буде значення <code>True</code> , якщо в операнда <code>x</code> значення <code>False</code> . Результатом виразу <code>not x</code> буде значення <code>False</code> , якщо в операнда <code>x</code> значення <code>True</code>



Нерідко на практиці використовується така операція, як логічне виключне або. Це — бінарна операція. Її результатом є істина, якщо операнди мають різні значення. Якщо операнди мають однакові значення, результатом операції виключного або є значення хиба. Іншими словами, логічне виключне або — це перевірка на предмет того, різні значення в операндів чи ні. За допомогою логічних операторів `or`, `and` і `not` операція виключного або для операндів `x` і `y` записується як `(x or y) and (not (x and y))`.

Окрім безпосередньо логічних значень можуть використовуватися й числові значення (та й не тільки). Якщо числове значення зустрічається в тому місці, де, за ідеєю, повинно бути значення логічного типу, починається інтерпретація нелогічного виразу як логічного. Інтерпретація виконується так: нульові числові значення інтерпретуються як `False`, а ненульові значення інтерпретуються як `True`.



Ситуація навіть цікавіша, ніж може здатися на перший погляд. Логічні значення `True` і `False` можна використати, відповідно, як числа 1 і 0. Наприклад, в арифметичних розрахунках замість 1 можемо використати `True`, а замість 0 — `False`. Однак до текстового формату (наприклад, передані як аргументи функції `print()`) значення `True` і `False` приводяться не як 1 і 0, а як "True" і "False". Про цю особливість слід пам'ятати.

Крім того, під час перевірки умов або обчислення логічних виразів можуть використовуватися не тільки безпосередньо логічні значення або числа, а й інші об'єкти. Як саме виконується «логічна» інтерпретація таких об'єктів ми дізнаємося, коли біжче познайомимося з методами ООП.

Програмний код, наведений у лістингу 1.5, дає уявлення про те, як, застосовуючи логічні оператори, обчислюють логічні вирази.

Лістинг 1.5. Логічні оператори

```
a=True
b=not a
print(a,b)
c=a and b
d=a or b
print(c,d)
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 1.5)

```
True False
False True
```



У цьому випадку змінна `a` посилається на логічне значення `True`. Значення змінної `b` – це значення виразу `not a`. Оскільки `a` – це `True`, то значення виразу `not a` дорівнює `False`. Тому результатом виразу `a and b` є значення `False`, а значенням виразу `a or b` є значення `True`.

Хоча за своєю суттю логічні оператори повинні повернати логічні значення, у Python це далеко не завжди так. Результатом виразів на основі операторів `or` і `and` є один із операндів відповідного виразу. А операнди не обов'язково повинні бути логічного типу — достатньо, щоб вони могли інтерпретуватися як логічні значення. Ситуацію ілюструє наступний приклад (програмний код в лістингу 1.6).

Лістинг 1.6. Знову логічні оператори

```
x=10      # Числова змінна
y=20      # Числова змінна
z=x and y # Логічне І
print(z)  # Результат логічного І
z=x or y # Логічне АБО
print(z)  # Результат логічного АБО
# Логічне заперечення
print(not x)
```

Phyton

У цьому випадку логічні оператори використовуються з числовими операндами. Після виконання програмного коду отримуємо такий результат:

Результат виконання програми (з лістингу 1.6)

```
20
10
False
```



Змінні `x` і `y` посилаються на цілочислові значення. Оскільки значення не нульові, то при виконанні логічних операцій обидві змінні інтерпретуються як такі, що мають значення `True`. Але інтерпретується як значення `True` і посилається на значення `True` — це далеко не одне й те саме.

Під час обчислення виразу `x and y` спочатку перевіряється значення першого операнда. Оскільки перший операнд `x` (значення 10) інтерпретується як `True`, результатом виразу повертається другий операнд — тобто `y` (значення 20). Аналогічно при обчисленні виразу `x or y`, оскільки перший операнд `x` інтерпретується як `True`, він же повертається як результат. Але результатом є реальне числове значення, на яке посилається змінна `x` (тобто значення 10).

Інша справа з оператором логічного заперечення `not`. Результатом виразу `not x` є значення `False`. Тобто в цьому випадку логічний оператор дає цілком «очікуваний» логічний результат.

Оператори порівняння дозволяють порівнювати на предмет рівності/нерівності різні значення. Зазвичай (хоча і не завжди) ідеється про числові значення. Результатом операції порівняння є логічні значення: `True`, якщо відповідне співвідношення правильне (відношення `e`), і `False`, якщо воно неправильне (відношення немає). Тут і далі ми говоримо про операції порівняння тільки, в основному, для числових значень. Оператори порівняння мови Python наведено в таблиці 1.6.

Таблиця 1.6. Оператори порівняння

Оператор	Опис
<	<i>Строго менше.</i> Результатом є True, якщо значення операнда ліворуч від оператора менше від значення операнда праворуч від оператора. Інакше повертається значення False
>	<i>Строго більше.</i> Результатом є True, якщо значення операнда ліворуч від оператора більше за значення операнда праворуч від оператора. Інакше повертається значення False
<=	<i>Менше або дорівнює.</i> Результатом є True, якщо значення операнда ліворуч від оператора не більше за значення операнда праворуч від оператора. Інакше повертається значення False
>=	<i>Більше або дорівнює.</i> Результатом є True, якщо значення операнда ліворуч від оператора не менше від значення операнда праворуч від оператора. Інакше повертається значення False
==	<i>Дорівнює.</i> Результатом є True, якщо значення операнда ліворуч від оператора дорівнює значенню операнда праворуч від оператора. Інакше повертається значення False
!=	<i>Не дорівнює.</i> Результатом є True, якщо значення операнда ліворуч від оператора не дорівнює значенню операнда праворуч від оператора. Інакше повертається значення False
is	Оператор <i>перевірки ідентичності</i> об'єктів. Як результат повертається значення True, якщо обидва операнди посилаються на один і той самий об'єкт. В іншому випадку (тобто якщо операнди посилаються на різні об'єкти) повертається значення False
is not	Оператор <i>перевірки неідентичності</i> об'єктів. Як результат повертається значення True, якщо операнди посилаються на різні об'єкти. В іншому випадку (тобто якщо операнди посилаються на один і той самий об'єкт) повертається значення False

Гадаемо, оператори порівняння особливих коментарів не потребують (може, тільки за винятком операторів перевірки ідентичності/неідентичності

Phyton
.....

об'єктів). Невеликі приклади використання операторів порівняння наведено в листингу 1.7.

Листинг 1.7. Оператори порівняння

```
a=100  
b=200  
print(a<b,a>=b,a==100,b!=199)
```

Результат виконання коду такий:

Результат виконання програми (з листингу 1.7)

```
True False True True
```



Якщо з порівнянням числових значень усе більш-менш зрозуміло, то оператори `is` та `is not` можуть бути не зовсім зрозумілими читачеві. Тут доречно згадати, що змінні в Python не містять значення, а посилаються на них. Ми говоримо про змінні, які посилаються на значення одного типу. Які при цьому можливі варіанти? По-перше, змінні можуть посылатися на різні значення. По-друге, змінні можуть посылатися на однакові значення (тобто фізично ці значення однакові, але кожне з них записано в пам'яті окремо). По-третє, змінні можуть посылатися не просто на однакові значення, а на одне й те саме значення. Друга і третя ситуації принципово різні. Причому якщо ми просто звертаємося до змінної, то фактично отримуємо те значення, на яке змінна посилається. При перевірці на предмет рівності (оператор `==`) або нерівності (оператор `!=`) значень, на які посилаються дві змінні, порівнюються значення, які повертаються за відповідними посиланнями. При цьому за допомогою зазначених операторів неможливо перевірити, реалізуються ці значення одним об'єктом чи різними об'єктами. Для виконання такої перевірки використовують оператори `is` та `is not`.

На закінчення пункту зробимо декілька важливих зауважень. Перше — щодо так званих *скорочених форм операціона присвоювання*.

Як ми вже знаємо, оператором присвоювання в Python є знак рівності `=`. Також існують так звані скорочені форми операціона присвоювання. Ідеється ось про що. Якщо необхідно виконати команду вигляду `x=x ⊕ y`, де через `x` і `y` позначені деякі змінні, а через `⊕` ми формально позначили один із арифметичних або побітових операцій, то цю команду можна

записати у вигляді $x \otimes =y$. Наприклад, замість команди $x=x+y$ можна використовувати команду $x+=y$.

Щодо самого оператора присвоювання (це друге зауваження), то в Python дозволено *багатократне і множинне* присвоювання. При багатократному присвоюванні в одному виразі використовується декілька операторів присвоювання. Прикладом такої ситуації може бути вираз $x=y=10$, яким змінним x і y присвоюється значення 10. Множинне присвоювання — це коли ліворуч від оператора присвоювання зазначено одразу декілька змінних. Як приклад такої ситуації може бути команда $a, b=1, 2$. У цьому випадку ліворуч від оператора присвоювання через кому вказано змінні a і b , а праворуч (теж через кому) — значення 1 і 2. У результаті змінна a отримує значення 1, а змінна b — значення 2. Принцип обробки такого роду виразів наступний: спочатку обчислюються значення праворуч від оператора присвоювання, а потім ці значення присвоюються змінним, зазначенним ліворуч від оператора присвоювання (кількість змінних ліворуч і кількість значень праворуч повинні збігатися). Тому, наприклад, щоб змінні a і b «обмінялися» значеннями, можна використати команду $a, b=b, a$.

Разом із тим, багатократне і множинне присвоювання слід використовувати вкрай обережно, оскільки під час роботи з такими даними, як, наприклад, *спісхи*, результати схожих операцій можуть бути цілком неочікуваними. Уся «складність» ситуації пов’язана, в основному, з тим, що в Python (як підкреслювалося і ще буде підкреслюватися) змінні не містять значення, а посилаються на них. Для деяких типів даних цей момент є принциповим. Усі подібні «премудрості» ми вивчатимемо в міру знайомства з різними типами даних.

Третє зауваження стосується пріоритету різних операторів. У складних виразах одночасно можуть бути присутніми найрізноманітніші оператори. Такі вирази обчислюються відповідно до пріоритету операторів. Спочатку обчислюються вирази й підвирази з операторами, які мають вищий пріоритет, а вже потім — із нижчим пріоритетом. Якщо декілька операторів мають однакові пріоритети, вираз обчислюється зліва направо. Коротко пріоритет операторів у Python такий (у порядку зменшення пріоритету):

- побітова інверсія, піднесення до степеня, знак числа (унарний мінус або унарний плюс);
- множення, ділення, ціличислове ділення, остача від ділення;
- додавання, віднімання;
- побітові зсуви;
- побітове *i*;
- побітове *виключне або*;
- побітове *або*;
- оператор присвоювання (включаючи і скорочені форми).



Вище згадувався знак числа — це унарний оператор, який ставиться перед числовим значенням і визначає, додатне число чи від'ємне. Для додатних чисел знак зазвичай не ставиться (така ось традиція), хоча ніхто не забороняє написати перед додатним числом знак +. Перед від'ємними числами ставиться знак -.

Для зміни порядку обчислення виразу можна використовувати круглі дужки. Більше того, наявність у правильному місці круглих дужок (навіть якщо крайньої потреби в них немає), не тільки робить код «надійнішим», а й значно покращує його сприйняття.

Далі ми детальніше обговоримо деякі особливості числових даних. Числові типи цікаві самі по собі й часто використовуються при створенні програмних кодів. Інші типи даних, такі як текст, списки, множини, кортежі й словники, ми розглянемо в наступних розділах книги.

Числові дані

Десять ночей у десять разів тепліші,
ніж одна. І в десять разів холодніші.

Л. Керром. «Аліса в Країні Див»

Числові значення можуть бути представлені декількома типами. Цілі числа реалізуються за допомогою типу `int`, дійсні нецілі числа (числа у форматі з плаваючою крапкою) реалізуються за допомогою типу `float`, а комплексні числа реалізуються у вигляді даних типу `complex`.



З математичної точки зору множина цілих чисел є підмножиною дійсних чисел, яка, у свою чергу, є підмножиною комплексних чисел.

Нагадаємо, що комплексні числа — це числа, які можна зобразити у вигляді $z = x + iy$, де дійсні числа x і y називаються відповідно дійсною та уявною частиною комплексного числа, а уявна одиниця i за визначенням така, що $i^2 = -1$. Операції з комплексними числами (множення, ділення, додавання й віднімання) виконуються як зі звичайними алгебраїчними виразами, тільки з поправкою на те, що $i^2 = -1$. Результатом додавання, віднімання, множення й ділення двох комплексних чисел є комплексне число.

Числом, комплексно спряженим до числа $z = x + iy$, називається число $\bar{z} = x - iy$ (отримується внаслідок заміни у вихідному числі i на $-i$).

Модуль $|z|$ комплексного числа z — це завжди число дійсне, яке дорівнює квадратному кореню з добутку цього числа на комплексно спряжене:

$$|z| = \sqrt{z \cdot \bar{z}} = \sqrt{x^2 + y^2}.$$

Тут нас, насамперед, цікавитимуть прийоми створення числових літералів. Про те, як виконуються числові розрахунки, йтиметься трохи згодом — принаймні, після того, як ми познайомимося з основними

інструкціями керування (такими, як умовний оператор і оператори циклу).

Зазвичай числові літерали набираються в десятковій системі — тобто в тій системі, яку ми використовуємо в повсякденному житті. Це десять цифр від 0 до 9 включно, за допомогою яких ми й уводимо в програмному коді потрібне нам ціле або дійсне число. Цілі числа набираються у вигляді послідовності цифр і, якщо треба, знака числа. Дійсні числа набираються практично так само, тільки в них є ціла й дробова частини, а як роздільник цілої і дробової частини використовуємо крапку.



Літерал — це фіксоване значення, яке не може бути змінено в програмі. Зазвичай під літералами розуміють числа й текст, які використовуються явно в програмному коді. Наприклад, у команді `x=1.5` через `x` позначено змінну, а `1.5` — це літерал (числовий). Інший приклад: у команді `name="Іван Іванович"` текст `"Іван Іванович"` є літералом, а `name` — це змінна.

Але, в принципі, у програмному коді для цілих чисел значення можна подавати не тільки в десятковій системі числення, а й у двійковій, вісімковій і шістнадцятковій.



Двійкову систему числення ми вже обговорювали. У вісімковій системі числа записуються за допомогою восьми цифр: від 0 до 7 включно. Якщо у вісімковій системі число має позиційне зображення $\overline{a_n a_{n-1} \dots a_1 a_0}$ (тут параметри a_0, a_1, \dots, a_n можуть набувати значення 0, 1, ..., 7), то це, насправді, означає, що йдеться про число, яке в десяткову систему переводиться за формулою $\overline{a_n a_{n-1} \dots a_1 a_0} = a_0 \cdot 8^0 + a_1 \cdot 8^1 + a_2 \cdot 8^2 + \dots + a_n \cdot 8^n$.

Аналогічна справа з шістнадцятковою системою числення. Тільки тепер використовується, як неважко згадатися, 16 «символів»: десять цифр від 0 до 9 і ще шість букв від `a` до `f` включно. Ці букви позначають числа від 10 до 15 (тобто буква `a` відповідає числу 10, буква `b` — числу 11 і так далі, аж до букви `f`, яка відповідає числу 15).

Якщо в шістнадцятковій системі число має зображення $\overline{a_n a_{n-1} \dots a_1 a_0}$ (але тепер параметри a_0, a_1, \dots, a_n можуть набувати значення 0, 1, ..., 9, `a`, `b`, ..., `f`), то в десятковій системі числення це число розраховується так: $\overline{a_n a_{n-1} \dots a_1 a_0} = a_0 \cdot 16^0 + a_1 \cdot 16^1 + a_2 \cdot 16^2 + \dots + a_n \cdot 16^n$.

Якщо літерали задаються не в десятковій системі числення, то починається вони повинні зі спеціального префікса, який «ідентифікує», до якої системи числення належить літерал. Для бінарних чисел (літерали у двійковій системі) префікс складається з нуля й букви `b` (великої або малої). Наприклад, літерал `0b101` означає число 5. Цілком законною буде, скажімо, команда `x=0b101`, якою змінній `x` присвоюється числове значення 5.



Якщо б нам знадобилося присвоїти змінній `x` значення -5 (тобто від'ємне значення), причому набрати літерал у двійковому коді, то відповідна команда виглядала б як `x=-0b101`. Тобто ми використовуємо знак «мінус» в літералі. Все, що ми говорили вище про бінарне кодування від'ємних чисел у комп'ютері, про принцип «доповнення до нуля» — у цьому випадку нам не потрібно. Чому? Тому що вище йшлося про механізм зображення від'ємних чисел у пам'яті комп'ютера. Тут ідеється про формальний запис числа з використанням двійкового коду. Все, що ми пишемо в програмному коді, призначено для того, хто буде цей код читати — тобто для програміста, користувача, читача. Отже, ніхто не забороняє нам використовувати в числовому літералі знак «мінус», навіть якщо цей літерал зображає число у двійковій системі числення. Комп'ютер, коли прийде його черга взятися за виконання програмного коду, автоматично переведе зазначений літерал у «правильне», з точки зору комп'ютера, зображення. Це саме зауваження (відносно знака числа) стосується й інших систем числення: вісімкової й шістнадцяткової.

Ознакою числа, записаного у вісімковій системі числення, є префікс `0o` (нуль і велика або мала буква `o`), а числа, записані в шістнадцятковій системі, починаються з префікса `0x` (нуль і велика або мала буква `x`). Наприклад, число 123 у вісімковій системі числення запишеться як `0o173`. Це саме число в шістнадцятковій системі зображається літералом `0x7b`.

Комплексні числа вводяться в «природному» форматі, тобто у вигляді суми дійсної й уявної частин. Ознакою уявності числа є суфікс `j` (велика або мала буква), який розташований одразу після числового значення. Іншими словами, якщо після числового літерала вказати (без пробілів або інших роздільників) букву `j`, то отримаємо уявне число. Наприклад, комплексне число $3 + 2i$ запишеться в програмному коді як `3+2j`. Уявна одиниця `i` у вигляді програмного коду реалізується як `1j` і так далі. Також для створення комплексного числа можемо скористатися функцією `complex()`. Першим аргументом функції передається дійсна частина

Python

комплексного числа, а другий аргумент — уявна частина комплексного числа. Так, число $3 + 2i$ можемо отримати за допомогою команди `complex(3, 2)`, а уявній одиниці i відповідає інструкція `complex(0, 1)`.

Для введення дуже великих або, напроти, дуже маленьких (за модулем) чисел зручно скористатися зображенням числа у вигляді **мантиси і показника степеня**. Як роздільник мантиси і показника степеня використовують букву e (велику або малу). Наприклад, числовий літерал $1.2e3$ позначає число $1,2 \cdot 10^3$, а числовий літерал $1.2e-5$ відповідає числу $1,2 \cdot 10^{-5}$.

Вище ми вже розглядали арифметичні й побітові оператори. З цілочисловими значеннями (дані типу `int`) можуть використовуватися й ті, й інші. З дійсними значеннями (дані типу `float`) використовуються арифметичні оператори. Також для виконання математичних розрахунків призначено цілий ряд вбудованих функцій. Деякі корисні в роботі математичні функції наведено в таблиці 1.7.

Таблиця 1.7. Деякі математичні функції

Функція	Опис
<code>abs()</code>	Обчислення модуля числа. Число, для якого обчислюється модуль, вказується аргументом функції. Може використовуватися з комплексними числами. Наприклад, результатом кожного з виразів <code>abs(5.0)</code> , <code>abs(-5.0)</code> і <code>abs(3+4j)</code> є значення <code>5.0</code>
<code>bin()</code>	Функція призначена для перетворення числа з десяткової системи числення у двійкову. Вихідне число вказується аргументом функції, а результатом є текстове зображення для двійкового коду числа. Наприклад, результатом виразу <code>bin(9)</code> буде текст <code>"0b1001"</code>
<code>complex()</code>	Функцію використовують для створення комплексних чисел на основі дійсної й уявної частин числа (переданих аргументами функції) або на основі текстового зображення комплексного числа. Наприклад, результатом виразів <code>complex(3, 4)</code> і <code>complex("3+4j")</code> буде комплексне число <code>3+4j</code>

Функція	Опис
float ()	Функцію використовують для перетворення числових значень і текстових зображень для дійсних чисел у числові значення типу float. Наприклад, результатом кожного з виразів <code>float (5), float ("5"), float ("5.") ifloat ("5.0")</code> є значення 5.0
hex ()	Функцію призначено для перетворення числа з десяткової системи числення в шістнадцяткову. Аргумент функції — число в десятковій системі числення. Результат функції — текстове зображення цього числа в шістнадцятковій системі. Наприклад, результатом виразу <code>hex (123)</code> буде текст "0x7b"
int ()	Функція для перетворення об'єкта (наприклад, тексту) у ціле число. Якщо аргументом функції передано дійсне число (тип float), то результат обчислюється відкиданням дробової частини в дійсному числі. Якщо аргументом вказати текстове зображення цілого числа, результатом буде саме це число. Причому число (у текстовому зображені) може бути не тільки в десятковій системі, а й у двійковій, вісімковій або шістнадцятковій. У цьому випадку другим аргументом передається ціле число, що визначає вихідну систему числення (відповідно, 2, 8 або 16). Наприклад, результатом кожного з виразів <code>int(123.4), int("123"), int("0b1111011",2), int("0o173",8) i int("0x7b",16)</code> є значення 123
max ()	Функція для визначення максимального значення з набору чисел. Наприклад, результатом виразу <code>max (-2, 4, 9, -1)</code> є значення 9
min ()	Функція для визначення мінімального значення з набору чисел. Наприклад, результатом виразу <code>min (-2, 4, 9, -1)</code> є значення -2
oct ()	Функція призначена для перетворення числа з десяткової системи числення у вісімкову. Аргументом указується число в десятковій системі числення, а результатом є текстове зображення цього числа у вісімковій системі. Наприклад, результатом виразу <code>oct (9)</code> буде текст "0o11"

Функція	Опис
pow ()	Функція піднесення числа до степеня. Якщо у функції два аргументи, то результатом є перше число в степені, що визначається другим числом. Іншими словами, результатом виразу $\text{pow}(x, y)$ є значення $x^{**}y$. Якщо у функції три аргументи, то після піднесення до степеня обчислюється остача від ділення на третій аргумент: тобто результатом виразу $\text{pow}(x, y, z)$ є значення $(x^{**}y) \% z$. Наприклад, результатом виразу $\text{pow}(2, 3)$ буде значення 8, а результатом виразу $\text{pow}(2, 3, 5)$ буде значення 3
round ()	Функція призначена для округлення дійсних значень до цілочислових. Аргументом указується значення, яке округлюється. Округлення виконується за такими правилами: <ul style="list-style-type: none"> якщо дробова частина значення, яке округлюється, менша від 0.5, округлення виконується до найближчого меншого цілого числа; якщо дробова частина значення, яке округлюється, більша за 0.5, округлення виконується до найближчого більшого цілого числа; якщо дробова частина значення, яке округлюється, дорівнює 0.5, округлення виконується до найближчого парного цілого числа. Якщо функції передати другий аргумент, то він визначатиме кількість знаків у дробовій частині, до яких буде виконуватися округлення (тобто в цьому випадку округлення виконується не до цілого числа, а до дійсного з кількістю розрядів після крапки, яка визначається другим аргументом функції). Наприклад, результатом виразу $\text{round}(4.6)$ (дробова частина 0.6) є значення 5, у виразу $\text{round}(-4.6)$ (дробова частина 0.4) — значення -5, у виразу $\text{round}(4.4)$ (дробова частина 0.4) — значення 4, у виразу $\text{round}(-4.4)$ (дробова частина 0.6) — значення -4, у виразу $\text{round}(4.5)$ (дробова частина 0.5) — значення 4, а у виразу $\text{round}(-4.5)$ (дробова частина 0.5) — значення -4. Для порівняння: результат виразу $\text{round}(1.23456, 3)$ — число 1.235

З деякими із цих функцій ми ще матимемо справу й познайомимося з ними більше при розв'язанні задач у наступних розділах книги.



Велика кількість математичних функцій стає доступною після підключення модуля `math`. Модулі обговоримо далі, проте, одразу зазначимо, що нічого складного в підключені модуля немає: в даному випадку в програму достатньо додати інструкцію `import math`. Після цього, наприклад, можемо в програмному коді використовувати тригонометричні функції, такі як `sin()` (синус), `cos()` (косинус), `tan()` (тангенс) і багато інших. Щоправда, для зазначеного способу імпортування модуля під час виклику функцій із цього модуля доведеться перед іменем функції вказувати називу модуля (розділювач імені модуля й імені функції – крапка): наприклад, `math.sin()`, `math.cos()` або `math.tan()`. Також у цьому модулі визначені ірраціональні сталі: $\pi \approx 3,14159265$ (інструкція `math.pi`) і $e \approx 2,7182818$ (інструкція `math.e`).

Трохи пізніше в книзі ми розглянемо приклади, в яких програмні коди, написані мовою Python, використовуються для розв'язання обчислювальних задач.

18. Підключення модулів

Робити їй було абсолютно нічого,
а сидіти без діла, самі знаєте, спра-
ва нелегка.

Л. Керролл. «Аліса в Країні Див»

Зазвичай під *модулем* розуміють деякий файл із програмою або блоком програмного коду. Під час написання великих програм не завжди зручно весь програмний код зберігати в одному файлі. Тому його розбивають на окремі частини або на модулі. Можливий інший варіант: у написанні власної програми ми хочемо використовувати частину програмного коду, який був створений раніше (і знаходиться в якомусь певному модулі). Щоб використовувати такий код, необхідно *імпортувати* відповідний модуль. Для імпортовання модулів застосовується інструкція `import`, а після неї вказується назва модуля, який підключають (імпортують). Ми будемо імпортувати вбудовані модулі Python — тобто ті модулі, які є складовою частиною середовища розробки мовою Python. Наприклад, якщо ми хочемо імпортувати математичний модуль `math` (модуль містить різні математичні функції та утиліти), то використовуємо інструкцію `import math`. Якщо імпортовані модулі декілька, то після інструкції `import` імена імпортованих модулів перераховуються через кому.

Після того, як модуль підключено, описані в ньому функції, змінні й інші корисні конструкції можна використовувати в програмному коді. Але при цьому кожного разу необхідно явно зазначати ім'я модуля, в якому описано змінну або функцію. Використовується так званий *крапковий синтаксис*: спочатку зазначаємо ім'я модуля, і через крапку ім'я змінної або назvu функції (з усіма належними аргументами). Наприклад, якщо ми хочемо використовувати змінну, яку описано в модулі, то відповідна інструкція буде виглядати як `модуль.змінна`. Причому

попередньо за допомогою команди `import` модуль необхідно імпортувати модуль. Більш конкретно, у модулі `math` є змінна `r` зі значенням сталої $\pi \approx 3,14159265$. Якщо ми хочемо побачити значення цієї змінної, то спочатку за допомогою команди `import math` підключаемо модуль `math`, а потім за допомогою команди `print(math.pi)` відображаємо значення змінної `r` з модуля `math`.

Замість того щоб використовувати ім'я модуля під час звернення до його «вмісту», можемо для модуля створити «псевдонім». Модуль підключаємо за допомогою команди у форматі `import` модуль `as` ім'я. Іншими словами, в інструкції підключення модуля після імені модуля через ключове слово `as` можна вказати ідентифікатор, який буде використовуватися замість імені модуля. Тобто модуль все одно підключається, але коли ми звертаємося до змінних і функцій цього модуля, то вказуємо не ім'я модуля, а ідентифікатор (той, який після ключового слова `as`). Наприклад, якщо ми підключаємо модуль за допомогою команди `import` модуль `as` ім'я, то звертатися до змінної з модуля потрібно у форматі ім'я.змінна. Якщо згадати про модуль `math` і підключити його за допомогою команди `import math as m`, то роздрукувати значення змінної `r` можна за допомогою команди `print(m.pi)`.



Якщо ми підключили модуль із «псевдонімом», то звертатися до вмісту модуля доведеться через «псевдонім» — спроба використовувати ім'я модуля призведе до помилки.

Впадає в око незручність, пов'язана з необхідністю вказувати в явному вигляді ім'я модуля (або його «псевдонім») при звертанні до змінних і функцій модуля. Але тут приховано й головну перевагу: безпосередньо в програмному коді ми можемо визначити змінну (функцію або щось інше) з таким самим іменем, як і в модулі, який підключається. У цьому випадку наявність або відсутність імені модуля при звертанні до змінної (або функції) дозволяє однозначно ідентифікувати, про яку програмну конструкцію йдеТЬся.

Разом із тим, можна підключати не весь модуль, а тільки деякі його утиліти (zmінні або функції). Скажімо, якщо з модуля нас цікавить тільки одна змінна, то можемо скористатися командою `from` модуль `import` змінна. Після цього змінну можна використовувати без посилання на ім'я модуля.

Phyton

Так, щоб використовувати в програмному коді змінну `pi` без зазначення перед її іменем назви модуля `math`, застосовуємо інструкцію імпорту `from math import pi`.

Якщо ми хочемо підключити всі утиліти з деякого модуля, корисною буде інструкція `from` модуль `import *` (тобто після інструкції `import` вказуємо зірочку `*`).

Тернарний оператор

Кращий спосіб пояснити – це самому зробити.

Л. Керром. «Аліса в Країні Див»

Часто виникає необхідність виконувати в програмному коді різні команди, залежно від того, чи є істинною певна умова. Взагалі така задача розв'язується за допомогою *умовного оператора*, з яким ми познайомилися в наступному розділі. Разом із тим, є «спрощена» форма умовного оператора, яку, за аналогією з мовами C++ і Java, можна було б назвати *тернарним оператором* (хоча, звичайно, конструкція, яка розглядається далі, різко контрастує зі звичними уявленнями про оператор).



Зазвичай оператори бувають *унарні* і *бінарні*. В унарного оператора є один операнд, у бінарного оператора — два операнди. Тернарний оператор — оператор, у якого три операнди. Один із операндів — це умова, яка перевіряється. Ще два операнди — значення, одне з яких повертається як результат, залежно від умови, яка перевіряється.

Тернарний оператор повертає значення. Причому це значення залежить від істинності або хибності певної умови. Шаблон тернарного оператора такий (жирним шрифтом виділено основні інструкції у виразі для тернарного оператора):

значення_1 **if** умова **else** значення_2

Тернарний оператор — досить складна конструкція. Спочатку вказується вираз (значення_1), який визначає значення тернарного оператора за істинності умови. Між цим виразом й умовою розміщують

Phyton

ключове слово `if`. Після умови слідує ключове слово `else`, а потім — вираз (значення `_2`), що визначає результат тернарного оператора, якщо умова хибна. Невеликий приклад використання тернарного оператора наведено в лістингу 1.8.

Лістинг 1.8. Тернарний оператор

```
# Зчитується перше число
a=float(input("Уведіть перше число: "))
# Зчитується друге число
b=float(input("Уведіть друге число: "))
# Перше значення
value_1="Перше число більше, ніж друге."
# Друге значення
value_2="Друге число не менше, ніж перше."
# Викликається тернарний оператор
res=value_1 if a>b else value_2
# Відображується результат
print(res)
```

Результат виконання цього програмного коду може бути таким (жирним шрифтом виділені введені користувачем значення):

Результат виконання програми (з лістингу 1.8)

```
Уведіть перше число: 12
Уведіть друге число: 30
Друге число не менше, ніж перше.
```

Або таким:

Результат виконання програми (з лістингу 1.8)

```
Уведіть перше число: 30
Уведіть друге число: 12
Перше число більше, ніж друге.
```

У цьому випадку ситуація досить проста. Спочатку користувач уводить два числа, які записуються у змінні `a` і `b`. Для зчитування введеного

Розділ 1. Програма мовою Python

користувачем значення ми застосовуємо функцію `input()`. Проте, оскільки ця функція як результат повертає введене користувачем значення в текстовому форматі, для надійності перетворюємо зчитане значення у формат числа з плаваючою крапкою, для чого передаємо результат виклику функції `input()` як аргумент функції `float()`.

Значення змінної `res` обчислюється на основі тернарного оператора. У тернарному операторі перевіряється умова `a < b`. Якщо умова істинна, то змінній `res` присвоюється текстове значення змінної `value_1`. Якщо умова хибна, то змінній `res` присвоюється значення змінної `value_2`. Значення змінної `res` відображується у вікні виводу за допомогою функції `print()`.

Резюме

– Не сумуй, – сказала Аліса. – Рано чи пізно все стане зрозуміло, все стане на свої місця і вишикується в єдину красиву схему, як мережива. Стане зрозуміло, навіщо все було потрібно, тому що все буде правильно.

Л. Керролл. «Аліса в Країні Див»

Підіб'ємо короткі підсумки цього розділу.

- Програма, написана на Python, — це послідовність команд. Для виконання цих команд використовується спеціальна програма-інтерпретатор.
- У програмі можуть використовуватися змінні. У Python змінна посилається на значення, а не містить його, як у багатьох інших мовах програмування.
- У Python існує декілька типів даних. Числові значення реалізуються даними типу `int` (цілі числа), `float` (дійсні числа) і `complex` (комплексні числа). Для позначення уявної частини комплексного числа використовують букву `j` (велику чи малу). Двійкові, вісімкові й шістнадцяткові літерали вводяться відповідно з префіксами `0b`, `0o` і `0x` (друга буква після нуля може бути великою або малою).
- Тексту відповідає тип `str`. Текстові літерали беруться у по-двійні (або одинарні) лапки.

- За допомогою типу `bool` реалізуються логічні значення. Дані цього типу можуть набувати значення `True` (істина) і `False` (хиба). Логічні значення є підвидом цілочислового типу даних, а отже, можуть використовуватися в арифметичних обчисленнях.
- Тип змінної явно зазначати не потрібно — він визначається на основі значення, на яке посилається змінна.
- Для введення даних із консолі використовують функцію `input()`, а для виведення — функцію `print()`.
- Для реалізації в програмному коді коментаря використовують символ `#`. Все, що знаходиться праворуч від цього символу, інтерпретатор ігнорує.
- Основні оператори Python можна розподілити на чотири групи: арифметичні, побітові, логічні й оператори порівняння.
- Арифметичні оператори: `+` (додавання), `-` (віднімання), `*` (множення), `/` (ділення), `//` (цилочислове ділення), `%` (остача від цілочислового ділення), `**` (піднесення до степеня).
- Побітові оператори: `~` (побітова інверсія), `&` (побітове *i*), `|` (побітове *aбо*), `^` (побітове *виключне або*), `<<` (побітовий зсув ліворуч), `>>` (побітовий зсув праворуч).
- Логічні оператори: `or` (логічне *або*), `and` (логічне *i*), `not` (логічне заперечення).
- Оператори порівняння: `>` (більше), `<` (менше), `<=` (не більше), `>=` (не менше), `==` (дорівнює), `!=` (не дорівнює). Також до операторів порівняння зазвичай відносять оператор перевірки ідентичності об'єктів `is` і оператор перевірки неідентичності об'єктів `not`.
- Оператор присвоювання має скорочені форми: наприклад, команду вигляду `x=x+y` можна записати у вигляді `x+=y`. Такого типу скорочені вирази можна використовувати для всіх арифметичних і побітових операторів.

Phyton

- Функція eval () дозволяє розрахувати вираз, який у вигляді тексту передано аргументом функції.
- Для підключення модулів використовується інструкція import, після якої вказується ім'я модуля, який підключають. Під час використання змінних (функцій) із підключенного модуля ім'я модуля (через крапку) зазначається перед іменем змінної (функції).
- Тернарний оператор має три операнди і повертає значення залежно від істинності або хибності певної умови (один із операндів тернарного оператора).



Розділ 2

Інструкції керування

Знаєш, одна з найсерйозніших
втрат у битві – це втрата голови.

Л. Керрол. «Аліса в Країні Див»

У даному розділі йтиметься про інструкції керування. До інструкцій керування у цьому випадку ми зараховуємо *умовний оператор* (із усіма його можливими модифікаціями), а також *оператори циклу* (у мові Python їх два). За великим рахунком, саме інструкції керування роблять програму справді програмою — тобто цілісною конструкцією, а не банальним набором послідовно виконуваних команд. Але про все по порядку. Насамперед, розглянемо умовний оператор, причому почнемо з найпростіших його версій.

Умовний оператор

Бачила я таку нісенітницю, в порівнянні з якою ця нісенітница – тлумачний словник!

Л. Керролл. «Аліса в Країні Див»

Умовний оператор (або, як його ще іноді називають, умовна інструкція) залежно від істинності чи хибності певної умови дозволяє виконувати різні блоки програмного коду. Загальна схема, відповідно до якої функціонує умовний оператор, виглядає так:

- Перевіряється деяка умова — зазвичай це вираз, значення якого може інтерпретуватися як істинне (значення `True`) або хибне (значення `False`).
- Якщо умова (вираз) інтерпретується як істинна, виконується виділена спеціальним чином послідовність команд.
- Якщо умова інтерпретується як хибна, виконується інша (але теж виділена спеціальним чином) послідовність команд.
- Після того, як одну або іншу послідовність команд умовного оператора виконано, керування передається тій команді, яка знаходиться після команди виклику умовного оператора.

Фактично, є два набори команд, а рішення про те, який із наборів команд виконати, приймається за результатами перевірки умови-виразу. Таким чином, у програмі створюється «точка розгалуження».

У мові Python умовний оператор реалізується у вигляді програмного блоку з наступним шаблоном (жирним шрифтом виділено ключові елементи):

```
if умова:  
    команди_1  
else:  
    команди_2
```

Після ключового слова `if` указується `умова` — вираз логічного типу (або який допускає інтерпретацію в термінах `True` і `False`). Після `умови` ставиться двокрапка (тобто `:`). Далі йде блок команд, які виконуються, якщо `умова` істинна (у шаблоні це `команди_1`). Блок команд виділяється за допомогою *відступів* (це стандартний спосіб виділення блоків команд у мові Python). У принципі, кількість відступів може бути довільною — головне, щоб дляожної команди блоку робилася одна й та сама кількість відступів. Але загальноприйнятим є *правило робити 4 відступи* (4 пробіли).

Після закінчення блоку команд (виконуваних за істинної `умови`) іде ключове слово `else` (і двокрапка `:`). Ключове слово `else` повинно бути на тому ж рівні (така ж кількість відступів або пробілів), що й ключове слово `if`. Далі — ще один блок команд (у шаблоні позначено як `команди_2`), які виконуються, якщо `умова` хибна.



У Python декілька команд можуть знаходитися в одному рядку. У цьому випадку команди розділяються крапкою з комою. Також команди можна розташовувати в одному рядку з ключовими словами `if` і `else`. Але це не той стиль, якого слід дотримуватися.

Схема виконання умовного оператора ілюструється на рис. 2.1.

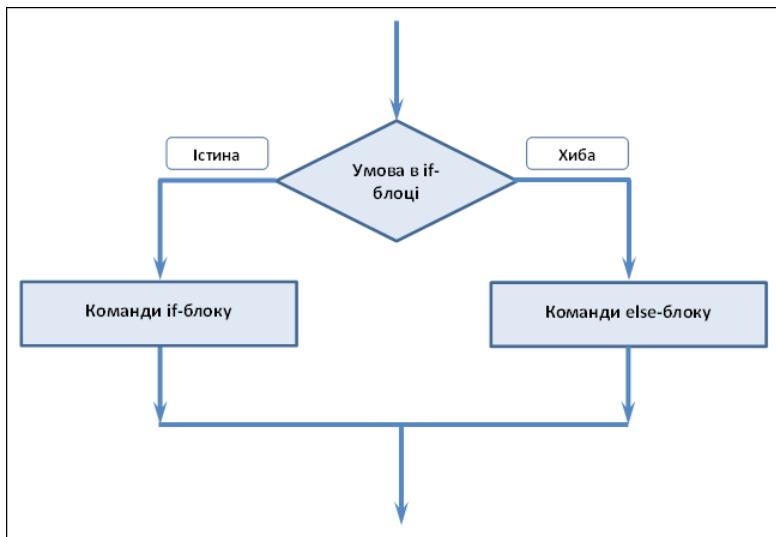


Рис. 2.1. Схема виконання умовного оператора

Приклад використання умовного оператора наведено в лістингу 2.1.

Лістинг 2.1. Умовний оператор

```

# Користувач уводить значення
res=eval(input("Уведіть що-небудь: "))
# Використовуємо умовний оператор для перевірки
# типу введеного користувачем значення
if type(res)==int:
    # Якщо ціле число
    print("Ви ввели ціле число!")
else:
    # Якщо щось інше
    print("Це точно не ціле число!")
# Після виконання умовного оператора
print("Роботу закінчено!")
  
```

Під час запуску програми на виконання спочатку з'являється запит на введення користувачем якогось значення. Уведене користувачем значення зчитується й запам'ятується. Потім перевіряється тип уведеного користувачем значення: якщо точніше, перевіряється, чи ввів користувач ціличислове значення. Для цього використовується умовний оператор.

Якщо користувач увів ціле число, виводиться відповідне повідомлення. Якщо те, що ввів користувач, не є цілим числом, теж з'являється повідомлення, але вже інше. Нарешті, після завершення виконання умовного оператора з'являється повідомлення про закінчення роботи програми.



Нагадаємо, що під час роботи з середовищем IDLE введення користувачем значення здійснюється через вікно інтерпретатора. Під час роботи з середовищем PyScripter для введення значення відображається спеціальне вікно з полем уводу.

Зрозуміло, що результат виконання програмного коду залежить від уведеного користувачем значення. Наприклад, якщо користувач уводить ціле число, результат може бути таким, як показано нижче (жирним шрифтом виділено введене користувачем значення):



Результат виконання програми (з лістингу 2.1)

Уведіть що-небудь: **12**

Ви ввели ціле число!

Роботу закінчено!

Якщо користувач уводить не ціле число або зовсім не число, результат буде дещо іншим (жирний шрифт — те, що вводить користувач):



Результат виконання програми (з лістингу 2.1)

Уведіть що-небудь: **12.0**

Це точно не ціле число!

Роботу закінчено!

Відмінність у тому, що в першому випадку користувач (за нашим бажанням) уводить значення 12, і це — ціле число. У другому випадку вводиться число 12.0, і це вже число дійсне — наявність десяткової крапки переворює цілочисловий літерал у літерал для числа з плаваючою крапкою (тип float).

Щодо безпосередньо програмного коду (див. лістинг 2.1), то деякі команди краще прокоментувати. Так, ми вже знайомі з функцією `input()`. Вона повертає текстове зображення тих даних, які вводить користувач.

Тобто навіть якщо користувач уводить число, воно все одно буде зчитане в текстовому форматі (це називається текстове зображення числа). Пікантність ситуації в тому, що у нас немає гарантії, що користувач увів число. У принципі, це може бути будь-який текст. Тому ми застосуємо хитрість: результат функції `input()` (а це, нагадаємо, введене користувачем текстове значення) передаємо як аргумент функції `eval()`. Функція `eval()` намагається «обчислити» вираз, «захований» у тексті. Якщо там заховане ціле число, результатом буде це число. Якщо дійсне число — результатом буде воно (дійсне число). Якщо просто якийсь текст, то він текстом і залишиться.



Під час уведення тексту його треба взяти в подвійні лапки. Якщо цього не зробити, виникне помилка, причому виникне вона на етапі виконання функції `eval()`. Тобто на запит увести значення треба реагувати акуратно: або вводимо число, або текст — але текст обов'язково в лапках!

Результат запам'ятовується у змінній `res`. Далі в гру вступає умовний оператор. За допомогою функції `type()` обчислюємо тип уведеного користувачем значення. Зокрема, в умовному операторі перевіряється вираз `type(res)==int`, який має значення `True`, якщо тип значення, на яке посилається змінна `res`, дорівнює `int` (тобто якщо це ціле число).

Якщо користувач дійсно ввів ціле число, виконується команда `print("Ви ввели ціле число!")`. В іншому випадку (якщо не виконано умову `type(res)==int`) реалізується команда `print("Це точно не ціле число!")`. На цьому блок умовного оператора закінчується. Команда `print("Роботу закінчено!")` виконується вже після завершення умовного оператора.

Також хочеться звернути увагу на одну важливу обставину: якщо користувач уведе з формальної точки зору не число, а вираз, який повертає цілочисловий результат (наприклад, `1+2**3-4`), ефект буде такий, як наче б користувач увів ціле число:



Результат виконання програми (з лістингу 2.1)

Уведіть що-небудь: **1+2**3-4**

Ви ввели ціле число!

Роботу закінчено!

Описаний вище умовний оператор зазвичай називають *if*-оператором або *if-else*-оператором. Разом із тим, цей оператор може використовуватися й у дещо іншому вигляді. Так, допускається використання *спрощеної форми* оператора без *else*-блоку. Шаблон використання умовного оператора в цьому випадку такий (жирним шрифтом виділено ключові елементи):

```
if умова:  
    команди
```

Як і в попередньому випадку, спочатку перевіряється умова, яка вказана після ключового слова *if*. Якщо умова істинна, виконується блок команд. Після цього керування передається команді, розміщенній після умовного оператора. Якщо умова хибна, то робота умовного оператора на цьому закінчується й виконується наступна команда після умовного оператора. Як виконується умовний оператор у спрощеній формі, ілюструє схема на рис. 2.2.

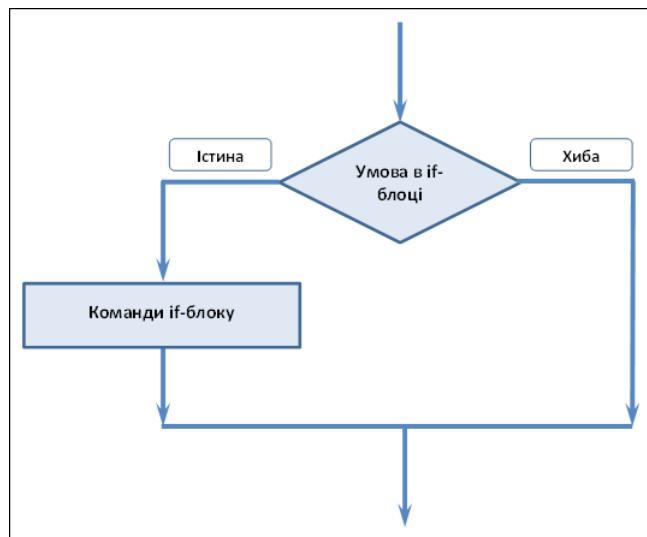


Рис. 2.2. Схема виконання умовного оператора у спрощеному вигляді без *else*-блоку

Невеликий приклад використання спрощеної версії умовного оператора наведено в лістингу 2.2.

Лістинг 2.2. Спрощена форма умовного оператора

```
# Користувач уводить значення
res=eval(input("Уведіть що-небудь: "))
# Тип значення запам'ятовуємо в змінній
resType=type(res)
# Використовуємо умовні оператори (спрощена форма)
# для перевірки типу введеного користувачем значення
if resType==int:
    # Якщо ціле число
    print("Це ціле число!")
if resType==float:
    # Якщо дійсне число
    print("Це дійсне число!")
if resType!=int and resType!=float:
    # Якщо не число
    print("Напевно, це текст!")
# Після виконання умовних операторів
print("Роботу закінчено!")
```

Ситуація дуже схожа на попередню, проте, є деякі відмінності. Головна полягає в тому, що тепер відслідковуються два типи даних: `int` і `float`. Перевірка виконується за допомогою трьох умовних операторів — кожний у спрощеній формі, йдуть один за іншим. У першому умовному операторі перевіряється умова `resType==int` (попередньо змінній `resType` присвоєно значення `type(res)`). Ця умова істинна, якщо змінна `res` посилається на ціличислове значення. Якщо це дійсно так, виконується команда `print("Це ціле число!")`. Якщо умова хибна, нічого не виконується (мається на увазі, що нічого не виконується першим умовним оператором).

У другому умовному операторі перевіряється умова `resType==float`. На випадок істинності такої умови передбачено команду `print("Це дійсне число!")`. Якщо умова хибна — на цьому етапі нічого не відбувається.

У третьому умовному операторі перевіряється умова `resType!=int and resType!=float`. Умова полягає в тому, що тип значення, на яке

Python

посиляється змінна `res`, не збігається з типом `int` і, одночасно, не збігається з типом `float`. Фактично, ця умова виконується, якщо не виконана жодна з двох умов, які використовувалися в попередніх умовних операторах. Так ось, у цьому випадку виконується команда `print("Напевно, це текст!")`.

Таким чином, є три умовних оператори, в кожному з яких перевіряється певна умова. Умови такі, що істинною може бути одна умова й тільки одна з них. За істинності тієї або іншої умови виводиться відповідне повідомлення. Нижче наведено приклад того, як може виглядати результат виконання програми, якщо користувач уводить ціле число (жирним шрифтом, як завжди, виділено введене користувачем значення):



Результат виконання програми (з лістингу 2.2)

Уведіть що-небудь: **12**

Це ціле число!

Роботу закінчено!

Якщо ввести дійсне число, результат виглядатиме так:



Результат виконання програми (з лістингу 2.2)

Уведіть що-небудь: **12.0**

Це дійсне число!

Роботу закінчено!

Нарешті, якщо ввести текст, отримаємо такий результат:



Результат виконання програми (з лістингу 2.2)

Уведіть що-небудь: **«Вивчаємо Python»**

Напевно, це текст!

Роботу закінчено!

Ще раз звертаємо увагу, що текст треба вводити в лапках. Причина пов'язана з тим, як ми оброблюємо введене користувачем значення — ми це значення передаємо аргументом функції `eval()`.

Що можна сказати про розглянутий вище програмний код? По-перше, він робочий (тобто виконується). По-друге, він виконується правильно: якщо ми вводимо ціле число, якщо ми вводимо дійсне число, і якщо ми вводимо щось інше (текст) — у кожному з цих випадків з'являється «персональне», призначене саме для цього випадку, повідомлення. Проте є одне «але». Полягає воно в тому, що, з одного боку, і ми це зазначали, виконуватися може лише одна з трьох умов, що перевіряються в умовних операторах, а з іншого боку, навіть якщо якусь умову виконано, умови, які залишилися, все одно будуть перевірятися. Більш конкретно, уявімо, що в першому ж умовному операторі умова виявилася істинною. Ця обставина означає, що дві інші умови однозначно хибні, але їх все одно буде перевірено. А це — зайві обчислення. І хоча на час виконання такого невеликого за обсягом програмного коду така надлишковість практично не вплине, певне незадоволення від ситуації все ж залишається. У такому випадку доцільніше використати декілька вкладених умовних операторів. І на цей випадок у Python є спеціальна версія умовного оператора.

Одна дуже корисна модифікація умовного оператора дозволяє послідовно перевіряти декілька умов. Якщо дивитися в корінь, то це, насправді, система вкладених умовних операторів. Використовується такий шаблон (жирним шрифтом виділено ключові елементи):

```
if умова_1:  
    команди_1  
elif умова_2:  
    команди_2  
elif умова_3:  
    команди_3  
...  
elif умова_N:  
    команди_N  
else:  
    команди
```

У цьому випадку початок традиційний: після ключового слова **if** вказується умова для перевірки (позначено **умова_1**). Якщо умова істинна, виконуються команди **if**-блоку (позначені як **команди_1**). Якщо вираз **умова_1** хибний, перевіряється умова, вказана після ключового слова **elif** (у шаблоні умову позначено як **умова_2**, після умови ставиться

двохрапка :). Якщо ця, друга, умова істинна, виконуються команди для відповідного elif-блоку (команди даного блоку позначені як *команди_2*), і на цьому роботу умовного оператора закінчено. Якщо ж і вираз *умова_2* хибний, перевіряється умова в наступному elif-блоці, і у випадку його істинності — команди цього блоку. Якщо під час перебору elif-блоків жодної істинної умови не виявлено, виконуються команди else-блоку, який розташований у шаблоні останнім. На рис. 2.3 представлена схема виконання умовного оператора з перевіркою декількох умов.

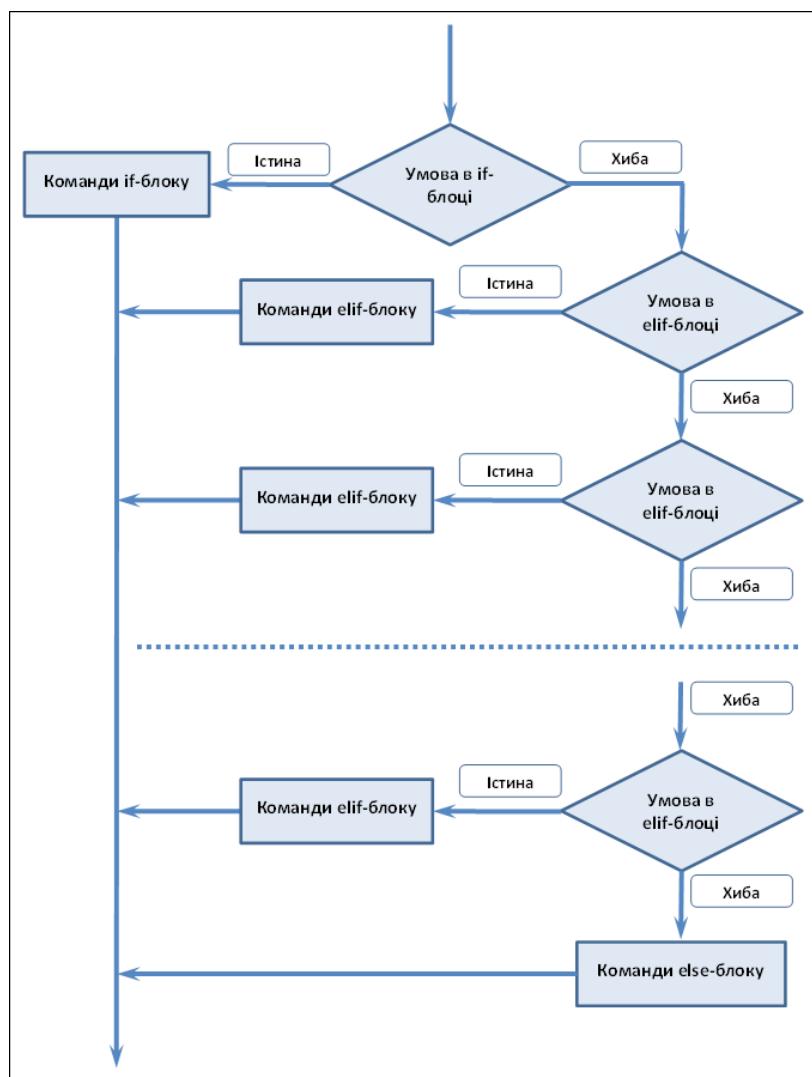


Рис. 2.3. Схема виконання умовного оператора з перевіркою декількох умов

У лістингу 2.3 наведено програмний код, у якому ідею попереднього прикладу реалізовано через один умовний оператор із перевіркою декількох умов.

Лістинг 2.3. Умовний оператор із перевіркою декількох умов

```
# Користувач уводить значення
res=eval(input("Уведіть що-небудь: "))
# Тип значення запам'ятовуємо в змінній
resType=type(res)
# Використовуємо умовний оператор (з декількома умовами)
# для перевірки типу введеного користувачем значення
if resType==int:
    # Якщо ціле число
    print("Це ціле число!")
elif resType==float:
    # Якщо дійсне число
    print("Це дійсне число!")
else:
    # Якщо не число
    print("Напевно, це текст!")
# Після виконання умовних операторів
print("Роботу закінчено!")
```

Результат виконання даного програмного коду достату такий самий, як і в попередньому прикладі, тому зупиняється на цьому не будемо. Однак алгоритм тут дещо інший. Так, насамперед, перевіряється умова `resType==int`. Якщо умова істинна, виводиться відповідне повідомлення, і робота умовного оператора закінчується. Це означає, що одразу буде виконано команду `print("Роботу закінчено!")` у кінці програми, а всі «внутрішні» умови (точніше, там залишилася одна умова) перевірятися не будуть.

Умова `resType==float` буде перевірятися лише в тому випадку, якщо хибна умова `resType==int`. А щодо `else`-блоку, то він виконується лише у тому випадку, якщо хибними є обидві умови.

Оператор циклу `while`

Як вона не намагалася, вона не могла знайти тут ні тіні сенсу, хоча всі слова були їй абсолютно зрозумілі.

Л. Керролл. «Аліса в Країні Див»

Оператори циклу дозволяють багаторазово виконувати визначений на-перед набір або групу команд. У мові Python є два оператори циклу. Спочатку ми познайомимося з оператором циклу, в якому використовується `while`-інструкція. Тому зазвичай такий оператор циклу називають *оператором циклу while*, або `while`-оператором.

Шаблон у цього оператора циклу досить тривіальний (жирним шрифтом виділено ключові елементи):

```
while умова:  
    команди
```

Принцип роботи оператора циклу дуже простий: насамперед, коли черга доходить до виконання цього оператора, перевіряється умова, яку вказано після ключового слова `while`. Якщо умова істинна, виконуються команди в тілі оператора циклу. Після виконання команд знову перевіряється умова. Якщо умова істинна, виконуються команди, а потім перевіряється умова і так далі — доти, поки під час перевірки умови не виявиться, що вона хибна. На цьому виконання оператора циклу закінчується, і виконується команда, наступна після умовного оператора. Схематично процес виконання оператора циклу проілюстровано на рис. 2.4.

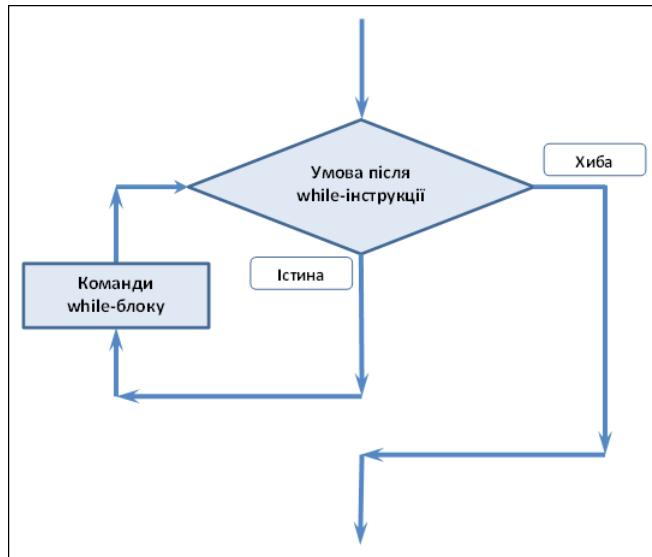


Рис. 2.4. Схема виконання оператора циклу

В оператора циклу є й «розширені» версія з ключовим словом `else`. Шаблон для оператора циклу в цьому випадку такий (жирним шрифтом виділено ключові елементи):

```
while умова:  
    команди_1  
else:  
    команди_2
```

Крім `while`-блоку в цьому випадку є ще й `else`-блок: після ключового слова `else` (і двокрапки `:`) розміщується блок команд, які виконуються тільки тоді, коли умова після інструкції `while` є хибною. Зокрема, на самому початку виконання оператора циклу перевіряється умова після інструкції `while`. Якщо умова хибна, виконуються команди в `else`-блокі, і на цьому робота оператора циклу закінчується. Якщо умова істинна, виконуються команди `while`-блоку. Після цього перевіряється умова. Якщо умова хибна, виконуються команди `else`-блоку, і закінчується робота оператора циклу. Якщо умова істинна, виконуються команди `while`-блоку, і знову перевіряється умова й так далі. Схему виконання такої «розширененої» версії оператора циклу зображенено на рис. 2.5.

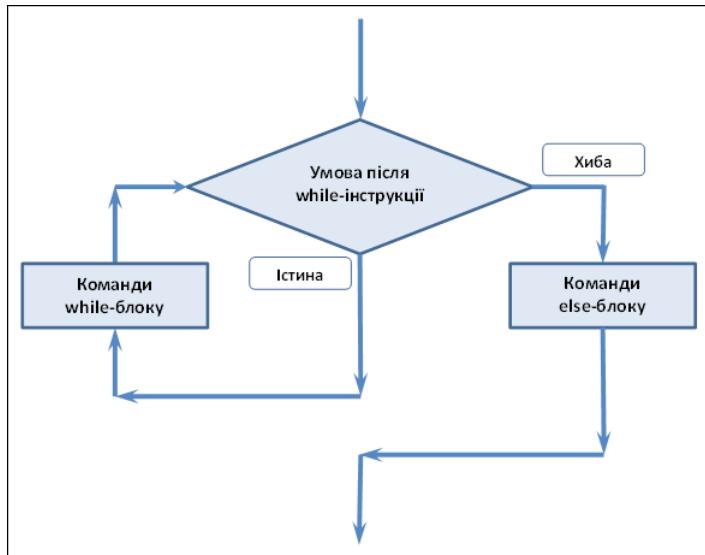


Рис. 2.5. Схема виконання оператора циклу з else-блоком

Якщо уважно проаналізувати цю схему, то нескладно зрозуміти, що команди else-блоку виконуються один і тільки один раз, причому на заключній стадії виконання оператора циклу. Такого ж ефекту можна досягти, якщо ці команди розташувати поза оператором циклу одразу після нього. Тому з else-блоку було б мало користі, якби не два **ключових слова** (две інструкції): `break` і `continue`. Інструкція `break` закінчує роботу оператора циклу без виконання else-блоку. Інструкція `continue` закінчує виконання поточного циклу й дозволяє перейти одразу до перевірки умови в while-блöці.

Далі ми проілюструємо використання оператора циклу `while` на декількох нескладних прикладах. Почнемо з прикладу, в якому обчислимо суму натуральних чисел. Відповідний програмний код наведено в лістингу 2.4.



Лістинг 2.4. Сума натуральних чисел і оператор циклу

```

print("Сума натуральних чисел")
n=100 # Кількість доданків
# Формуємо текст для відображення результату
text="1+2+...+"+str(n)+" ="
# Ітераційна змінна для оператора циклу

```

```
i=1
# Змінна для запису суми
s=0
# Оператор циклу для обчислення суми
while i<=n:
    # Додаємо доданок до суми
    s=s+i
    # Змінюємо ітераційну змінну
    i=i+1
# Відображуємо результат
print(text,s)
```

Результат виконання цього програмного коду наведено нижче:

Результат виконання програми (з лістингу 2.4)

Сума натуральних чисел

1+2+...+100 = 5050

Основу програмного коду складає оператор циклу, під час виконання якого здійснюються основні обчислення. Перед виконанням оператора циклу команда `print("Сума натуральних чисел")` виводить повідомлення про те, що ми збираємося обчислювати суму натуральних чисел. Далі оголошується декілька змінних. Змінна `n` (зі значенням 100) визначає кількість доданків у сумі. У цьому випадку обчислюватимемо суму 100 натуральних чисел: тобто суму від 1 до 100. Також нам знадобиться текстове значення, яке ми виведемо в консоль під час відображення результату обчислень. З цією метою змінній `text` присвоюється значення `"1+2+...+"+str(n)+" ="`. Це — текстове значення. Отримуємо його шляхом об'єднання (конкатенації) трьох текстових фрагментів: `"1+2+...+", str(n) і " ="`. Для об'єднання використано оператор додавання. Питання можуть виникнути з другим «фрагментом», який у цьому випадку представлено інструкцією `str(n)`. Результатом даного виразу є текстове зображення числового значення, на яке посилається змінна `n`. Для переведення числового значення в текстове ми використали функцію `str()`, яка й призначена для таких цілей.

Також ми визначаємо ітераційну (використовується в операторі циклу для «індексації» циклів) змінну `i` з початковим значенням 1. Це перше

значення, яке ми додамо до суми. Результат (значення суми) будемо записувати в змінну s , у якої початкове значення нульове. Після цього виконується оператор циклу. Починається він з інструкції `while`, після якої вказано умову $i <= n$. Це означає, що оператор буде виконуватися доти, поки значення змінної i (ітераційна змінна, вона ж визначає доданок до суми) не перевищує значення змінної n (останній доданок у сумі). У тілі оператора циклу виконуються дві команди. Командою $s=s+i$ додається черговий доданок до суми, а потім командою $i=i+1$ на одиницю збільшується значення ітераційної змінної. Після виконання оператора циклу командою `print(text,s)` результат обчислень відображається в консольному вікні.



Замість команди $s=s+i$ можна було використати еквівалентну їй команду $s+=i$, а замість команди $i=i+1$ – відповідно, команду $i+=1$.

Специфіка оператора циклу така, що в командах циклу повинна бути заладена принципова можливість для зміни (у результаті виконання цих команд) умови, яка перевіряється за кожний цикл. Інакше, можемо отримати нескінчений цикл. Хоча, з іншого боку, навіть формально нескінчений цикл ще не означає, що код створено неправильно. У лістингу 2.5 наведено приклад програмного коду з нескінченим, на перший погляд, оператором циклу (який, звичайно ж, насправді не є нескінченим).



Лістинг 2.5. Оператор циклу з break-інструкцією

```
print("Сума натуральних чисел")
n=100 # Кількість доданків
# Формуємо текст для відображення результату
text="1+2+...+"+str(n)+" ="
# Ітераційна змінна для оператора циклу
i=1
# Змінна для запису суми
s=0
# Оператор циклу для обчислення суми
while True:
    # Додаємо доданок до суми
    s+=i
    # Змінюємо ітераційну змінну
```

```
i+=1
if i>n:
    break
# Відображаємо результат
print(text,s)
```

Результат виконання цього програмного коду абсолютно такий, як і в попередньому прикладі. Тобто у цьому випадку ми маємо справу практично з тією ж задачею і розв'язуємо її тими ж методами. Просто ми трохи інакше організували оператор циклу. А саме — після ключового слова `while` як умову вказано значення `True`. Це — константа. Які б команди в тілі оператора циклу не виконувалися, умова не зміниться. Тому, щоб вийти з оператора циклу в потрібний момент (закінчити роботу оператора циклу), у тілі оператора ми розмістили умовний оператор, у якому перевіряється умова `i>n`. Якщо умова істинна, виконується інструкція `break`. Виконання цієї інструкції приводить до завершення роботи оператора циклу, що нам, власне, й потрібно.



У першому прикладі з оператором циклу умова `i<=n` була умовою **продовження** роботи оператора. У другому прикладі умова `i>n` в умовному операторі є умовою **закінчення** роботи оператора циклу. Неважко помітити, що якщо істинна умова `i<=n`, то хибна умова `i>n`, і навпаки.

Наступний приклад, який ми розглянемо — задача про обчислення площин фігури, обмеженої двома кривими. Тут, у цьому прикладі, ми використовуємо вкладені оператори циклу (тобто один оператор циклу викликається в тілі другого оператора циклу).

Щодо безпосередньо розв'язуваної задачі, то нам доведеться обчислити площину фігури, яка обмежена двома кривими, рівняння яких $y(x) = x$ і $y(x) = x^2$. Графіки цих кривих наведено на рис. 2.6.

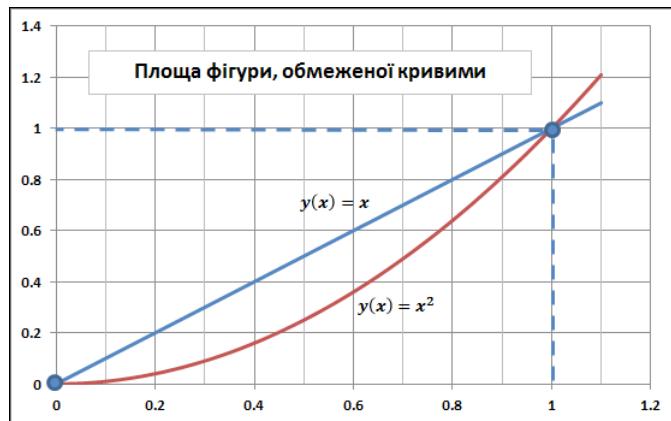


Рис. 2.6. Обчислення площи фігури, обмеженої двома кривими

Це — пряма лінія й парабола. Криві (лінія $y(x) = x$ задля загальності викладу розглядається як крива) перетинаються у двох точках: у точці $x = 0, y = 0$ і в точці $x = 1, y = 1$ (точки визначаються як розв'язок рівняння $x = x^2$). Виходить така собі «пелюстка», площею якої нам і треба буде обчислити.



Задача має точний розв'язок. А саме, площа S зазначеної фігури

$$S = \int_0^1 (x - x^2) dx = \frac{1}{6} \approx 0,166667.$$

Однак, ми використовуємо дещо інший

підхід. Метод, який описується далі й застосовується нами для обчислення площи фігури, має відношення до теорії ймовірностей і математичної статистики (зазвичай, такий підхід називають методами Монте-Карло).

При обчисленні площи фігури ми будемо виходити з таких міркувань. По-перше, ми бачимо, що область, обмежена кривими, повністю потрапляє в одиничний квадрат (без дотримання відповідного масштабу по координатних осіях) із лівою нижньою вершиною в точці початку координат і правою верхньою вершиною в точці з одиничними координатами. Якщо ми випадковим чином виберемо точку всередині цього квадрата, то вона з певною ймовірністю потрапить в область, обмежену кривими (тобто потрапить усередину «пелюстки»). Теорія ймовірностей стверджує, що ця сама ймовірність дорівнює відношенню площи «пелюстки» і квадрата. У квадрата з одиничною стороною площа дорівнює одиниці. Тому площа «пелюстки» (яку нам необхідно обчислити) дорівнює ймовірності, з якою випадковим чином вибрана точка потрапляє всередину «пелюстки». Це буде «по-друге».

По-третє, якщо взяти дуже багато випадкових точок, рівномірно розподілених по квадрату, і обчислити відношення кількості точок усередині «пелюстки» до загальної кількості точок, в ідеалі отримаємо оцінку, близьку до ймовірності потрапляння випадково выбраної точки всередину «пелюстки».

Ми всю цю процедуру трохи модифікуємо і зробимо так. Замість того, щоб генерувати випадкові точки, покриємо весь квадрат рівновіддаленими вузловими точками. Порахуємо, скільки їх потрапило всередину «пелюстки» (тобто області, обмеженої кривими), і поділимо на загальну кількість точок. Це й буде результат. Програмний код, у якому реалізовано такий підхід, наведено в лістингу 2.6.

Лістинг 2.6. Обчислення площини фігури й оператори циклу

```
# Кількість рівних інтервалів, на які діляться
# сторони однічного квадрата
n=500
# "Ціна поділки" - відстань між сусідніми точками
dz=1/n;
# Кількість точок, які потрапляють усередину області
pts=0
# Початкове значення індексу, що визначає стовпчик точок
i=0
# Зовнішній оператор циклу. Перебираємо стовпчики точок
while i<=n:
    # x-координата точки
    x=dz*i
    # Початкове значення другого індексу
    # для точок стовпчика
    j=0
    # Внутрішній оператор циклу. Перебираємо точки
    # в одному стовпчику
    while j<=n:
        # y-координата точки
        y=dz*j
        # Умовний оператор: перевіряємо, чи потрапила точка
        # всередину області
```

Phyton

```
if y<=x and y>=x**2:  
    # Ще одна точка всередині області  
    pts=pts+1  
    # Значення другого індексу збільшуємо на одиницю  
    j=j+1  
    # Значення першого індексу збільшуємо на одиницю  
    i=i+1  
# Обчислюємо площину фігури  
s=pts/ (n+1) **2  
# Відображаємо результат  
print("Площа фігури:",s)
```

Результат, який наведено нижче, досить близький до точного розв'язку:



Результат виконання програми (з лістингу 2.6)

Площа фігури: 0.16694355799379285

Щоб зрозуміти логіку обчислень, можна подумки уявити, як ми розбиваємо кожну зі сторін квадрата на певну кількість інтервалів. Кількість цих інтервалів записується у змінну n (значення 500). Границі інтервалів будемо називати *вузловими точками*. Через кожну вузлову точку на сторонах квадрата проводимо горизонтальній вертикальні лінії. Точки перетину цих ліній — це саме ті точки, які нам потрібні. Кожну таку точку можна «ідентифікувати» за допомогою двох індексів. Перший індекс визначає вузлову точку по горизонталі, а другий — вузлову точку по вертикалі. На перетині ліній, що проходять через ці вузлові точки, знаходиться «ідентифікована» точка всередині квадрата. Якщо ми зафіксуємо перший індекс i будемо брати різні значення для другого індексу, то всі відповідні точки будуть знаходитися на одній вертикальній прямій. Про такі точки будемо говорити, що вони знаходяться в одному стовпчику. Якщо зафіксувати другий індекс i і брати різні значення першого індексу, то всі відповідні точки будуть знаходитися на одній горизонтальній прямій. Про такі точки можемо говорити, що вони формують ряд точок. У кожному ряду й у кожному стовпчику розміщено рівно $n+1$ точок (якщо враховувати й ті точки, що знаходяться на координатних осіах).

Відстань (по горизонталі або по вертикалі) між двома сусідніми вузловими точками дорівнює, очевидно, одиниці, поділеній на кількість

інтервалів, на які розбивалася кожна зі сторін квадрата: така «щіна по-длки» записується в змінну `dz` (значення $1/n$). У змінну `pts` (початкове значення 0) будемо записувати кількість точок, які потрапили всередину «пелюстки».

Початкове значення індексу `i`, що визначає стовпчик точок, повинне бути нульовим (нульовий індекс відповідає точці на координатній осі). Через перший індекс, нагадаємо, «нумеруються» стовпці точок. Коли запускається зовнішній оператор циклу, в ньому перевіряється умова $i \leq n$. Тому цикл виконується доти, поки значення індексної змінної `i` не перевищить значення змінної `n`. У тілі оператора циклу командою `x=dz*i` обчислюється координата `x` (уздовж горизонтальної осі — *абсциса*) для точок, що знаходяться в цьому стовпчику (стовпчик, нагадаємо, визначається значенням індексу `i`). Також, оскільки далі ми плануємо перебирати точки в стовпчику, за допомогою команди `j=0` установлюємо початкове нульове значення для другого індексу (що визначає положення внутрішніх точок). Після цього виконується другий, внутрішній оператор циклу. У ньому перевіряється умова $j \leq n$ — тобто другий індекс `j` буде збільшуватися (про це ми дізнаємося пізніше) доти, поки він не перевищить граничне значення `n`. У тілі внутрішнього оператора циклу командою `y=dz*j` обчислюється координата `y` для точки в стовпчику (координата вздовж вертикальної осі — *ордината*). За допомогою умовного оператора перевіряємо, чи потрапляє точка всередину «пелюстки», і якщо так, то командою `pts=pts+1` на одиницю збільшується значення змінної `pts` (у яку, нагадаємо, записується кількість точок, що потрапляють усередину «пелюстки»).



В умовному операторі перевіряється умова `y <= x` and `y >= x**2`. Це і є умова потрапляння точки з координатами `x` (змінна `x`) і `y` (змінна `y`) всередину області, обмеженої кривими $y = x$ і $y = x^2$. Щоб точка потрапляла в цю область, необхідне одночасне виконання двох умов. По-перше, точка повинна знаходитися нижче прямої $y = x$, а це можливо, якщо $y \leq x$ (нестрога нерівність — якщо ми допускаємо, щоб точка могла опинитися не тільки нижче прямої, а й безпосередньо на прямій). По-друге, точка повинна знаходитися вище параболи $y = x^2$ (або на самій параболі — такий варіант ми теж допускаємо). Відповідна умова виглядає як $y \geq x^2$. Отже, повинно виконуватися співвідношення $x^2 \leq y \leq x$. Якщо перекласти це на мову Python, то отримаємо `y <= x` and `y >= x**2`. До речі, замість інструкції `y <= x` and `y >= x**2`

цілком законно можна було використовувати вираз $x^{**2} <= y <= x$. Такого типу вирази в Python допустимі.

Перед закінченням внутрішнього оператора циклу командою `j=j+1` другий індекс збільшується на одиницю. Це остання команда внутрішнього оператора циклу. Внутрішній оператор циклу є передостанньою «командою» зовнішнього оператора циклу. А остання команда зовнішнього оператора циклу — це інструкція `i=i+1`, якою на одиницю збільшується значення першого індексу.

Після виконання зовнішнього оператора циклу змінна `pts` містить значення кількості точок, які потрапляють усередину "пелюстки". Загальна кількість точок, як неважко здогадатися, дорівнює $(n+1)^{**2}$ (у кожному з $n + 1$ стовпчиків по $n + 1$ точок — усього $(n + 1)^2$ точок). Тому, якщо ми поділимо одне значення на інше, отримаємо площину "пелюстки". Відповідне значення розраховується командою `s=pts / (n+1)**2`. Нарешті, командою `print("Площа фігури:", s)` відображається результат.

Крім оператора циклу `while`, у мові Python є ще один оператор циклу, який, відштовхуючись від ключового слова, що входить у вираз для цього оператора, називають оператором циклу `for`.

20.12.21

Т
↓

Оператор циклу `for`

Мало хто знаходить вихід, деякі не бачать його, навіть якщо знайдуть, а багато хто навіть не шукають.

Л. Керролл. "Аліса в Країні Див"

Порівняно з оператором циклу `while`, оператор циклу `for` можна було б назвати більш «спеціалізованим». З одного боку, оператор дуже гнуний і ефективний. З іншого, його успішне використання може викликати певні труднощі, так би мовити, ідеологічного характеру. Простіше кажучи, під час роботи з оператором `for` (залежно від режиму його використання) виникає багато всіляких нюансів. Тому ми поки що розглянемо найпростіші й найрозуміліші схеми застосування оператора циклу `for`, а вже по ходу книги будемо розширювати й вдосконалювати наші знання й навички з цього питання. Шаблон використання даного оператора такий (жирним шрифтом виділено ключові елементи):

for елемент **in** послідовність:
команди

Після ключового слова `for` указують деякий елемент (якщо простіше, то назву змінної), який послідовно набуває значення з послідовності (упорядкований набір значень). Між елементом і послідовністю — ключове слово `in`. Закінчується ця синтаксична конструкція двокрапкою `:`. Потім іде блок команд оператора циклу. В операторі циклу команди виділяються відступами — традиційно чотирма.



Часто (але не завжди) як послідовність в операторі циклу `for` використовують **списки**. Зі списками ми ще не знайомі. Основні відомості про списки (як і про інші типи даних, що містять колекції значень і підпадають під визначення послідовності) наведено в наступних розділах. Тут нас списки цікавлять тільки в контексті використання їх в операторі циклу `for`. Важливо знати, що список — це набір упорядкованих елементів (оформлених відповідним чином). Причому елементи можуть бути різного типу. Щоб створити список, достатньо у квадратних дужках через кому перерахувати елементи списку. Наприклад, конструкція `[1, 2, 3, 4, 5]` є списком із п'яти натуральних чисел.

Як послідовність можна використовувати текстові значення. У цьому випадку перебираються букви в тексті.

Виконується оператор циклу так: елемент (змінна, вказана перед ключовим словом `in`) набуває першого значення в послідовності, зазначеній після ключового слова `in`. Після цього виконуються команди оператора циклу. Потім елемент набуває другого значення з послідовності, і знову виконуються команди оператора циклу. І так далі, поки не буде закінчено перебiranня всіх значень у послідовності.



У блоці команд оператора циклу можна використовувати інструкції `break` і `continue`. Ефект такий самий, як і під час використання цих інструкцій в операторі циклу `while`: виконання інструкції `break` приводить до завершення оператора циклу, а виконання інструкції `continue` приводить до завершення поточного циклу й переходу до наступного.

В операторі циклу `for` можна використовувати `else`-блок. Шаблон оператора циклу `for` із `else`-блоком виглядає так (жирним шрифтом виділено ключові елементи):

```
for елемент in список:
    команди_1
else:
    команди_2
```

Команди в `else`-блоці виконуються, фактично, після завершення оператора циклу, як би ми його розуміли без `else`-блоку. Весь «виграш» від `else`-блоку в тому, що якщо робота оператора циклу закінчується внаслідок виконання інструкції `break`, то команди в блоці `else` не виконуються.

Роботу оператора циклу `for` проілюструємо на нескладних прикладах. У першу чергу, розглянемо приклад, у якому обчислюється сума натуральних чисел. Вище ми такий приклад розглядали, але використовували там оператор циклу `while`. Тут удається до послуг оператора циклу `for`. Ще одне важливe нововведення — використання функції `range()`, за допомогою якої ми будемо створювати *віртуальну* послідовність натуральних чисел. Щоб створити такий об'єкт, як віртуальна послідовність чисел, аргументом функції `range()` передаються перше число віртуальної послідовності та останнє число послідовності плюс один. Наприклад, якщо нас цікавить послідовність чисел від 1 до 5 включно, то для створення такої послідовності можемо скористатися інструкцією `range(1, 6)`. Таким чином, останнє число в послідовності на одиницю менше за другий аргумент функції `range()`.



Якщо функції `range()` передати тільки один аргумент, то сформована послідовність буде починатися з нуля. Якщо функції `range()` передати три аргументи, то третій аргумент буде визначати крок арифметичної прогресії: перший аргумент визначає початкове числове значення в послідовності, а кожне наступне отримуємо додаванням кроку прогресії (третій аргумент) — але не більше, ніж значення другого аргументу. Можна також сказати, що якщо третій аргумент не вказано, то, за замовчуванням, це значення дорівнює одиниці, а якщо не вказано перший аргумент, то його значення, за замовчуванням, нульове.

Ще одне зауваження стосується терміна *віртуальний*. Чому ми говоримо про числову послідовність, сформовану функцією `range()`, як про віртуальну? Справа в тому, що насправді функція повертає деякий об'єкт, який допускає з собою таке поводження, як наче б це була послідовність чисел (потім ми більш детально познайомимося з подібними об'єктами — коли в рамках концепції ООП будемо обговорювати *ітератори, функції-генератори* та *ітераційні об'єкти*). Проте, якщо ми присвоїмо результат функції `range()` деякій змінній і потім захочемо перевірити значення цієї змінної, послідовності

ми не побачимо — буде формальне позначення результату з посиланням на назву функції `range()`. Щоб наочно побачити, що ж «заховано» всередині, можна скористатися функцією формування списків `list()`. Як аргумент цій функції передається результат виклику функції `range()`. Наприклад, результатом виразу `list(range(1, 6))` є список `[1, 2, 3, 4, 5]`. Однак, звертаємо увагу читача, що для використання функції `range()` в операторі циклу `for` ніяких додаткових дій щодо "візуалізації" віртуальної послідовності робити не потрібно.

У лістингу 2.7 наведено програмний код, у якому сума натуральних чисел обчислюється за допомогою оператора циклу `for`.

Лістинг 2.7. Обчислення суми чисел

```
print("Сума натуральних чисел")
n=100 # Кількість доданків
# Формуємо текст для відображення результату
text="1+2+...+"+str(n)+" ="
# Змінна для запису суми
s=0
# Оператор циклу для обчислення суми
for i in range(1, n+1):
    # Додаємо доданок до суми
    s=s+i
# Відображаємо результат
print(text,s)
```

Після виконання цього програмного коду ми отримуємо такий результат:

Результат виконання програми (з лістингу 2.7)

```
Сума натуральних чисел
1+2+...+100 = 5050
```

Порівняно з прикладом із лістингу 2.4 зміни мінімальні й стосуються вони, в основному, оператора циклу. Так, оскільки в операторі циклу `for` змінна `i` «пробігає» значення із послідовності чисел від 1 до `n` (послідовність створюється виразом `range(1, n+1)`), то відпадає необхідність присвоювати змінній `i` початкове значення. Також немає потреби й у команді

збільшення на одиницю значення ітераційної змінної і в операторі циклу: перебiranня значень виконується автоматично. Тому без усяких додаткових команд змінна і «пробігає» значення від 1 до n включно.



Суму можна обчислити за допомогою вбудованої функції `sum()`. Як аргумент функції передається список елементів, суму яких обчислюється. Так, суму натуральних чисел від 1 до n (якщо значення цієї змінної задано) можемо обчислити командою `sum(range(1, n+1))`. Тут як аргумент функції `sum()` передається віртуальна послідовність `range(1, n+1)` — також можна робити.

Наступний приклад ілюструє, як в операторі циклу `for` використовують текст як послідовність для перебiranня значень. Якщо конкретніше, то ми беремо за основу текст і потім за допомогою оператора циклу `for` роздруковуємо текст по буквах з невеликими текстовими вставками. Тепер звернімося до програмного коду в лістингу 2.8.

Лістинг 2.8. Текст в операторі циклу

```
# Текст для оператора циклу
txt="Python"
# Змінна для нумерації букв
i=1
# Оператор циклу
for s in txt:
    # Формуємо допоміжний текст
    t=str(i)+"-а буква:"
    # Виводимо повідомлення
    print(t,s)
    # Змінюємо номер букви
    i=i+1
# Команда після закінчення оператора циклу
print("Роботу програми закінчено!")
```

У змінну `txt` записується базовий текст, який ми плануємо «перебирати» по буквах в операторі циклу. Також ми оголошуємо з початковим однічним значенням змінну `i`. Ми під час виведення букв із тексту `txt`

збираємося відображати номер кожної букви, і для цього нам знадобилася окрема змінна. Але окремо підкresлимо: хоча в операторі циклу ця змінна й використовується, перебiranня елементів послідовності (букв у тексті) виконується за допомогою іншої змінної — ми її назвали `s`. Інструкція `s in txt` в операторі циклу після ключового слова `for` означає, що змінна `s` буде послідовно набувати буквного значення на основі тексту змінної `txt`. Тому про змінну `s` ми можемо думати як про чергову букву в тексті `txt`. Причому перебираються букви строго в тій послідовності, як вони знаходяться в тексті.

В операторі циклу за допомогою команди `t=str(i)+"-а буква:"` ми формуємо й записуємо в змінну `t` допоміжний текст, який виходить унаслідок об'єднання текстового зображення порядкового номера `i` і букви в тексті й тексту `"-а буква:"`. Для переведення числового значення в текст використано функцію `str()`.

У результаті виконання команди `print(t,s)` у вікні виводу (в одному рядку) з'являється допоміжний текст із номером букви й сама буква. Після цього командою `i=i+1` на одиницю збільшується номер для наступної букви.

Після закінчення оператора циклу командою `print("Роботу програми закінчено!")` виводиться фінальне повідомлення. Нижче наведено результат виконання програми:

Результат виконання програми (з лістингу 2.8)

```
1-а буква: Р
2-а буква: у
3-а буква: т
4-а буква: h
5-а буква: о
6-а буква: н
Роботу програми закінчено!
```

Наступний приклад ілюструє використання інструкції `break` і блоку `else` в операторі циклу `for`. У програмі перевіряється тип елементів списку. Ми шукаємо текстові елементи. Для цього використовуємо оператор циклу. В операторі циклу послідовно перебираються елементи

списку, і їхній тип перевіряється на предмет того, текст це чи ні. Якщо в списку немає текстових елементів, програма виводить повідомлення відповідного змісту. Але якщо під час перебирання елементів текст усеж таки зустрічається, то виконання оператора циклу закінчується, і програма повідомляє користувачеві, що в списку серед елементів є текст. Програмний код, у якому реалізовано цю ідею, наведено в лістингу 2.9.

Лістинг 2.9. Оператор циклу з `else`-блоком

```
# Починаємо перевірку списку
print("Перевіряємо вміст списку:")
# Список для перевірки. Тексту не містить.
# Під час перевірки альтернативного списку слід
# помітити як коментар наступний рядок
myList=[1,3+2j,True,4.0]
# Альтернативний список із текстом.
# Під час перевірки цього списку слід
# відмінити коментар для
# наступного рядка (і видалити пробіл)
# myList=[1,3+2j,"Python",4.0]
# Відображаємо вміст списку
print("Список:",myList)
# Оператор циклу для перевірки типу елементів списку
for s in myList:
    # Перевіряємо елемент на "текстовість"
    if type(s)==str:
        # Якщо елемент текстовий
        print("У списку є текстові елементи!")
        # Закінчується виконання оператора циклу
        break
    # Блок else оператора циклу.
    # Виконується, тільки якщо не виконувалася
    # інструкція break
else:
    # Відображається повідомлення про відсутність у
    # списку текстових елементів
    print("У списку немає текстових елементів!")
    # Повідомлення про закінчення перевірки
```

Phyton

```
print("Перевірку закінчено.")
```

У програмі команда `myList=[1, 3+2j, True, 4.0]` оголошує список `myList`, який ми й будемо перевіряти на наявність у ньому текстових елементів (у нашому випадку, очевидно, таких немає, але ніхто не забороняє нам змінити список). Аби було видно, який саме список перевіряється, за допомогою команди `print("Список:", myList)` виводимо вміст списку в консольне вікно. Після цього запускається на виконання оператор циклу. Змінна `s` «пробігає» набір значень елементів зі списку `myList`. Для кожного значення змінної `s` в умовному операторі виконується перевірка виразу `type(s)==str`. Вираз істинний, якщо тип значення, на яке посилається змінна `s`, дорівнює `str` (тобто якщо змінна посилається на текстове значення). У цьому випадку команда `print("У списку є текстові елементи!")` виводить повідомлення про наявність у списку текстових елементів, а потім інструкція `break` закінчує роботу оператора циклу. Якщо вираз `type(s)==str` хибний, то нічого цього не відбувається, і починається наступний елемент списку.

Якщо під час перебирання елементів списку жоден із них не виявився текстовим, основна частина оператора циклу закінчується «нічим»: нічого не відбувається. Але це ще не закінчення оператора циклу. У нього є `else`-блок, який «вступає у гру», якщо в тілі оператора циклу не виконувалася інструкція `break`. В `else`-блоці виконується всього одна команда `print("У списку немає текстових елементів!")`. Отже, якщо `break`-інструкція не виконувалася (а це означає, що в списку немає текстових елементів), то з'явиться повідомлення з `else`-блоку. Якщо `break`-інструкція виконувалася, повідомлення з `else`-блоку не з'явиться. Зате в цьому випадку з'явиться повідомлення з умовного оператора (виконується перед `break`-інструкцією).

Результат виконання програми для випадку, коли в списку, що перевіряється, немає текстових елементів, показано нижче:



Результат виконання програми (з лістингу 2.9)

Перевіряємо вміст списку:

Список: [1, (3+2j), True, 4.0]

У списку немає текстових елементів!

Перевірку закінчено.

Якщо в списку текстові елементи є, результат виконання програми може бути таким:

Результат виконання програми (з лістингу 2.9)

Перевіряємо вміст списку:

Список: [1, (3+2j), 'Python', 4.0]

У списку є текстові елементи!

Перевірку закінчено.



Для зручності у вихідному програмному коді є дві команди оголошення списку myList, але тільки одну реалізовано у вигляді коментаря. У першому випадку список не містить текстових елементів, у другому — містить. За необхідності одну команду можна виділити коментарем, в іншій коментування відмінити.

Також звертаємо увагу й нагадуємо читачеві, що текст у Python можна відділяти як подвійними, так і одинарними лапками. Під час виведення вмісту списку з текстовим елементом текст відображається в одинарних лапках, хоча в програмному коді для цієї мети ми використовували подвійні лапки.

У наступному прикладі ми розв'яжемо задачу про пошук елементів, які збігаються у двох списках. Розв'язуючи її, ми скористаємося двома операторами циклу for, причому в одному операторі циклу буде викликатися інший — тобто йдеться про вкладені оператори циклу. Розгляньмо програмний код у лістингу 2.10.

Лістинг 2.10. Збіг елементів у списках

```
print("Пошук елементів, які збігаються.")
# 1-й список
A=[2,False,9.1,2-1j,"hello",5,"Python"]
# 2-й список
B=[5,3,"HELLO",7,12.5,"Python",True,False]
# Відображаємо вміст 1-го списка
print("1-й список:",A)
# Відображаємо вміст 2-го списка
print("2-й список:",B)
print("Збігаються:")
```

Phyton

```
# Індекс для нумерації елементів 1-го списку
i=0
# Зовнішній оператор циклу.
# Перебираємо елементи 1-го списку
for a in A:
    # Новий індекс елемента з 1-го списку
    i=i+1
    # Індекс для нумерації елементів 2-го списку
    j=0
    # Внутрішній оператор циклу.
    # Перебираємо елементи 2-го списку
    for b in B:
        # Новий індекс елемента з 1-го списку
        j=j+1
        # Умовний оператор. Перевіряємо
        # рівність елементів
        if a==b:
            # Якщо елементи збігаються
            txt=str(i)+"-й елемент із 1-го списку і "
            txt=txt+str(j)+"-й елемент із 2-го списку"
            print(txt)
# Закінчення програми
print("Пошук закінчено.")
```

У програмі оголошуються два списки, A і B, їхні елементи зазначають явно. Вміст списків відображається в консольному вікні. Крім списків, ми використовуємо дві змінні. Змінна i призначена для запам'ятовування індексів елементів із першого списку, а змінна j служить тій самій меті, але тільки стосовно другого списку. Змінна i одразу отримує початкове нульове значення. Це відбувається саме перед початком виконання зовнішнього оператора циклу, в якому змінна a «пробігає» значення зі списку A. Тобто в зовнішньому циклі перебираються елементи першого списку.

В операторі циклу команда `i=i+1` задає значення порядкового номера для елемента першого списку, який передбачається порівнювати з елементами другого списку. Також змінна `j` отримує нульове значення (ця змінна визначає порядковий номер елемента у другому списку). Перебирання елементів другого списку здійснюється у внутрішньому операторі циклу. У цьому внутрішньому операторі циклу змінна `b` послідовно набуває значення зі списку `B`.

У внутрішньому операторі циклу команда `j=j+1` на одиницю збільшує значення змінної `j`, так щоб першому елементу в списку відповідав перший порядковий номер. Після цього в умовному операторі перевіряється умова `a==b`. Виконання цієї умови означає, що елементи списків збігаються. У цьому випадку дві команди `txt=str(i)+"-й елемент із 1-го списку i "` та `" i txt=txt+str(j)+"-й елемент із 2-го списку"` формують текст для відображення, а командою `print(txt)` цей текст відображається в консольному вікні (вікні виводу). Таким чином, якщо елементи списків збігаються, з'являється повідомлення про те, які (за порядковим номером) елементи в першому й другому списках збігаються. Результат виконання цієї програми виглядає так:

Результат виконання програми (з лістингу 2.10)

Пошук елементів, які збігаються.

1-й список: [2, False, 9.1, (2-1j), 'hello', 5, 'Python']

2-й список: [5, 3, 'HELLO', 7, 12.5, 'Python', True, False]

Збігаються:

2-й елемент із 1-го списку і 8-й елемент із 2-го списку

6-й елемент із 1-го списку і 1-й елемент із 2-го списку

7-й елемент із 1-го списку і 6-й елемент із 2-го списку

Пошук закінчено.



Вище в програмному коді текст (zmінна `txt`) для відображення в консольному вікні за збігу елементів списків формувався в декілька етапів із дуже простої причини: одна команда виглядала б досить громіздко.

Із наведеного прикладу видно, що під час порівняння текстових змінних має значення не тільки буквений склад тексту, а й реєстр букв (великі чи малі).

Також варто зауважити, що в нашій програмі не передбачено обробки ситуації, коли збігів у списках немає. Нічого страшного в цьому разі не станеться, але результат буде виглядати не дуже естетично.

Хоч умовні оператори й оператори циклу є достатньо потужним засобом і дозволяють вирішувати чимало нетривіальних задач, у Python існує ще одна синтаксична конструкція, яка у певному сенсі може бути зарахована до інструкцій керування — у всякому разі, один із можливих варіантів її використання зводиться до організації точок розгалуження в програмі. Йдеться про обробку *виняткових ситуацій*.

Обробка виняткових ситуацій

– Не можна повірити в неможливе!
– Просто у тебе мало досвіду, –
зауважила Королева. – У твоєму
віці я приділяла цьому півгодини
щодня! В інші дні я встигала по-
вірити в десяток неможливостей
до сніданку!

Л. Керролл. «Аліса в Країні Див»

Який би витончений програмний код не створювався, оминути всі «небезпечні місця» і передбачити всі можливі варіанти розвитку ситуації виходить далеко не завжди. Особливо це актуально, коли в програмі частина даних зчитується з файла або вводиться в програму користувачем уже в процесі виконання програми. У цих випадках велика ймовірність отримання програмою некоректних даних. У подібних ситуаціях під час виконання програми можуть виникати помилки. Найголовніший і найнеприємніший наслідок виникнення помилки — це передчасне «аварійне» закінчення програми. У Python існує механізм, який дозволяє програмі виконуватися навіть після того, як виникла помилка. Усе це називається загальним і містким терміном: *обробка виняткових ситуацій*.



Помилки, або виняткові ситуації, бувають різні. Зазвичай виділяють три типи помилок:

- Помилки, пов’язані з неправильним синтаксисом команд. Іншими словами, це помилки, пов’язані з неправильно набраним кодом. Такі помилки виявляються просто й без особливих зусиль: при спробі запуску

на виконання програми з неправильними командами, швидше за все, буде виведено повідомлення про місце помилки й причини її виникнення.

- Помилки, пов’язані з «неправильним» алгоритмом виконання програми. Тобто помилки, пов’язані з тим, що програму як таку створено неправильно (хоча з формальної точки зору всі команди коректні). Це дуже «підступні» помилки, оскільки зазвичай жодних попереджувальних повідомлень не з’являється, програма працює, а результат – неправильний. Можливе вирішення проблеми: тестування програми на модельних прикладах або задачах, для яких відомий правильний результат.
- Помилки часу виконання, що виникають у процесі виконання програми й пов’язані з некоректністю переданих у програму даних, недоступністю ресурсів тощо.

Зрозуміло, що далеко не для кожної помилки можна здійснити обробку так, аби програма продовжувала виконуватися. Однак, у деяких випадках така обробка можлива. Власне про такі потенційно «відловлювані» помилки, їх перехоплення й обробку йтиметься далі.

Загальна ідея, покладена в основу методу обробки виняткових ситуацій, така: програмний код, у якому теоретично може виникнути помилка, спеціально виділяється — образно кажучи, «береться на контроль». Якщо під час виконання цього програмного коду помилка не виникає, то нічого особливого не відбувається. Якщо під час виконання «контрольованого» коду виникає помилка, то виконання коду зупиняється, і автоматично створюється **виняток** — об’єкт, який містить опис помилки, що виникла. З практичної точки зору, ми можемо думати про виняток, як про певне повідомлення, яке генерується інтерпретатором через помилку, що виникла під час виконання коду або через якісь інші обставини, близькі за своєю природою до помилки виконання коду. Хоча помилка й виняток — це не одне й те саме (виняток є наслідком помилки), ми, якщо це не буде призводити до непорозумінь, будемо ототожнювати ці поняття.

Для обробки виняткових ситуацій у мові Python використовують конструкцію `try-except`. Існують різні варіації використання цієї конструкції. Ми почнемо з найбільш простих ситуацій.

Можемо вчинити так: після ключового слова `try` і двокрапки розміщуємо блок програмного коду, який ми підозрюємо в можливій наявності помилки. Цей код будемо називати **основним** або **контрольованим**. По закінченні цього блоку розміщується ключове слово `except` (із двокрапкою),

після якого йде ще один блок програмного коду. Цей код називатимемо **допоміжним** або **кодом обробки винятку**. Тобто шаблон такий (жирним шрифтом виділено ключові елементи):

```
try:
    # основний код
except:
    # допоміжний код
```

Якщо під час виконання основного коду в блоці `try` помилка не виникла, то допоміжний програмний код в блоці `except` виконуватися не буде. Якщо під час виконання основного коду в блоці `try` виникла помилка, то виконання коду `try`-блоку припиняється, і виконується допоміжний код у блоці `except`. Після цього керування передається наступній команді після конструкції `try-except`.

Приклад використання конструкції `try-except` в описаному вище форматі наведено в лістингу 2.11. Йдеться про розв'язання лінійного рівняння вигляду $ax = b$. Це рівняння має очевидний розв'язок $x = b/a$ за умови, що параметр a відмінний від нуля. Оскільки параметри a і b уводить користувач, то, як кажуть, можливі варіанти.

Лістинг 2.11. Обробка виняткових ситуацій

```
print("Розв'язуємо рівняння ax = b")
# Початок try-блоку (основний код)
try:
    # Визначаємо перший параметр. Можлива помилка
    # перетворення тексту в число
    a=float(input("Уведіть a: "))
    # Визначаємо другий параметр. Можлива помилка
    # перетворення тексту в число
    b=float(input("Уведіть b: "))
    # Розв'язання рівняння. Можлива помилка
    # ділення на нуль
    x=b/a
    # Відображається значення для кореня рівняння.
    # Команда виконується, якщо до цього не виникли
```

Phyton

```
# помилки
print("Розв'язок рівняння: x =", x)
# Початок ехсерт-блоку (допоміжний код)
except:
    # Команда виконується, тільки якщо раніше
    # під час виконання основного коду виникла помилка
    print("Ви ввели некоректні дані!")
# Команда виконується після блоку try-except
print("Дякуємо, роботу програми закінчено.")
```

Результат виконання цього програмного коду залежить від того, яке значення вводить користувач — у тому числі, при некоректно введенному значенні може виникнути помилка, і такі ситуації оброблюються в програмному коді. Конкретніше, на початку виконання програми користувачу пропонують увести значення для змінних `a` і `b` (команди `a=float(input("Уведіть a: "))` і `b=float(input("Уведіть b: "))`). У цьому випадку введені користувачем значення перетворюються у числовий формат.



Для перетворення текстового представлення числа в числове значення (у форматі з плаваючою крапкою) використано функцію `float()`.

Зрозуміло, що якщо користувач уведе неправильне значення (тобто не число), то виникне помилка. Тому відповідні команди розміщено в блоці `try`. У цьому ж блоці є команда `x=b/a`, якою на основі значень змінних `a` і `b` присвоюється значення змінній `x`. До цієї команди «черга» дійде, тільки якщо будуть успішно виконані попередні команди з присвоювання значень змінним `a` і `b`. Але навіть коли все пройшло гладко на попередніх етапах, нас може чекати новий сюрприз: якщо змінній `a` присвоєно нульове значення (ніхто не забороняє користувачеві так зробити), то під час виконання команди `x=b/a` виникає помилка ділення на нуль. Тому цю команду також поміщено в блок `try`. У цьому блоці є ще одна команда — інструкція `print("Розв'язок рівняння: x =", x)`, яка відображає значення для кореня рівняння.

Окрім блоку `try`, у програмному коді є блок `except`. У цьому блоці лише одна команда `print("Ви ввели некоректні дані!")`. Блок

except «вступає у гру», тільки якщо в блоці try виникла помилка. Іншими словами, команди в блоці try виконуються одна за одною, і поки не виникає помилка, усе відбувається так, як наче б вони були звичайнісінькими командами, без усякого блоку try. Якщо помилка так і не виникла, після закінчення виконання блоку try команди в блоці except не виконуються, а одразу починают виконуватися команди після конструкції try-except. У цьому разі це команда `print("Дякуємо, роботу програми закінчено.")`. Якщо на якомусь етапі в блоці try з'явилася помилка, виконання команд блоку try припиняється, і починают виконуватися команди (у нашому прикладі вона одна) у блоці except. Потім виконуються команди після конструкції try-except (тобто остання команда в програмному коді виконується в будь-якому випадку — незалежно від того, була помилка в try-блоці чи ні).

Нижче показано, як може виглядати результат виконання програми у випадку, якщо всі дані користувач уводить коректно (тут і далі введені користувачем значення виділено жирним шрифтом):

Результат виконання програми (з лістингу 2.11)

Розв'язуємо рівняння $ax = b$

Уведіть a : **5**

Уведіть b : **8**

Розв'язок рівняння: $x = 1.6$

Дякуємо, роботу програми закінчено.

Якщо замість числового значення ввести дещо інше, можемо отримати зовсім інший результат:

Результат виконання програми (з лістингу 2.11)

Розв'язуємо рівняння $ax = b$

Уведіть a : **text**

Ви ввели некоректні дані!

Дякуємо, роботу програми закінчено.

Майже такий самий результат отримуємо, якщо для змінної a вказати нульове значення:

 **Результат виконання програми (з лістингу 2.11)**

```
Розв'язуємо рівняння ax = b
Уведіть a: 0
Уведіть b: 3
Ви ввели некоректні дані!
Дякуємо, роботу програми закінчено.
```

У принципі, описана вище схема обробки виняткових ситуацій багато в чому проста й зручна. Проте її вона не позбавлена недоліків. У першу чергу, одразу впадає в очі, що різні помилки оброблюються однаково. Тобто, яка б не виникла помилка, реакція програми буде однакова.



Більше того, ситуація може бути ще більш неоднозначною. Припустімо, ми випадково (чи не дуже) помилилися в написанні назви, наприклад, функції `float()` в команді `a=float(input("Уведіть a: "))`. Під час запуску на виконання програмного коду отримаємо таку послідовність повідомлень:

```
Розв'язуємо рівняння ax = b
Ви ввели некоректні дані!
Дякуємо, роботу програми закінчено.
```

Чому так відбувається? Тому що у використаній нами схемі переходяться всі помилки, і тому що програмний код Python виконується інтерпретатором рядок за рядком (без попередньої компіляції). Коли черга доходить до виконання команди з невідомою функцією (неправильна назва функції), звичайно ж виникає помилка (класу `NameError`), і ця помилка переходиться в блоці `except`. Як наслідок, ми навіть не зможемо ввести значення для змінної `a`.

З іншого боку, якщо йдеться не про неправильну назву функції (наприклад, `float()`), а про некоректну інструкцію (припустімо, подвійні лапки в тексті не поставлено), то це вже неправильний синтаксис команди, про що з'явиться повідомлення одразу після запуску програми. Зазвичай, такі помилки автоматично виділяються у вікні редактора кодів ще під час набору. Хоча, звичайно, багато чого залежить від можливостей редактора.

Більш витончена обробка виняткових ситуацій передбачає її більш індивідуальний, так би мовити, підхід. Йдеться про те, щоб обробка помилок базувалася на типі або характері помилки. Зрозуміло, це можливо. Причому описана вище схема обробки виняткових ситуацій зазнає

мінімальних змін. Виглядає це так: у конструкції `try-except` після блоку `try` розміщують декілька `except`-блоків, причому для кожного блоку явно вказують тип помилки, яка оброблюється в цьому блоці. Ключове слово, що визначає тип помилки, ставиться після ключового слова `except` відповідного блоку.



Під час виникнення помилки генерується виняток і створюється об'єкт, що описує помилку. Тому тип помилки — це, насправді, **клас**, на основі якого створюється об'єкт винятку (більш точна назва, прийнята в Python, — **екземпляр винятку**). Іншими словами, «типологія» помилок базується на ієрархії класів. Кожен клас відповідає певній помилці. Про роботу з класами й об'єктами (екземплярами класу) поговоримо трохи пізніше. Та й особливості використання класів й об'єктів тут нас мало цікавлять. У всякому разі, не буде великою проблемою, якщо ми будемо інтерпретувати назву класу помилки просто як певне ключове слово, що визначає тип, якому належить помилка.

У цьому випадку шаблон програмного коду, що містить обробку помилок різного типу, має такий вигляд (жирним шрифтом виділено ключові елементи):

```
try:
    # основний код
except Тип_помилки_1:
    # допоміжний код
except Тип_помилки_2:
    # допоміжний код
...
except Тип_помилки_N:
    # допоміжний код
```

Цей код реалізується так. Виконуються команди `try`-блоку. Якщо виникла помилка, то виконання команд `try`-блоку припиняється і починається послідовний перегляд `except`-блоків на предмет збігу типу помилки, що виникла, і типу помилки, зазначеного після ключового слова в `except`-блоці. Щойно збіг знайдено, виконуються команди відповідного `except`-блоку, після чого керування переходить до команди після конструкції `try-except`.



Збіг типів під час пошуку потрібного `except`-блоку для обробки помилки не обов'язково повинен бути «букальним». Річ ось у чому.

Ми вже знаємо, що тип помилки — це насправді клас, який описує цю помилку. Але у класів можуть бути підкласи (похідні класи). Це приблизно так, якби в типу були підтипи. Або, іншими словами, для деяких категорій помилок існує більш детальна класифікація порівняно з іншими помилками. Наприклад, клас `ZeroDivisionError`, який відповідає помилці ділення на нуль, є підкласом класу `ArithmeticError` (арифметична помилка). А ще в класі `ArithmeticError` є підкласи `FloatingPointError` (помилка під час операції із плаваючою точкою) і `OverflowError` (помилка перевовнення). Так ось, під час обробки виняткових ситуацій перехоплюються не тільки помилки певного (зазначеного в `except`-блоці) класу, а й помилки підкласів цього класу. Наприклад, якщо ми в `except`-блоці вкажемо клас помилки `ZeroDivisionError`, то будуть перехоплюватися помилки типу `ZeroDivisionError` (ділення на нуль). Але якщо в `except`-блоці вказати клас `ArithmeticError`, то в цьому блоці будуть перехоплюватися й оброблятися помилки типу `ZeroDivisionError`, `FloatingPointError` і `OverflowError`.

Якщо під час перебирання `except`-блоків збіг (за типом помилки) не знайдено, виконання коду припиняється і з'являється повідомлення про помилку. Правда, можуть використовуватися вкладені конструкції `try-except`. У цьому випадку необроблена помилка може перехоплюватися зовнішнім `except`-блоком (але така можливість повинна бути передбачена в програмі).

Якщо під час виконання `try`-блоку помилок не було, коди в `except`-блоках не виконуються.

У лістингу 2.12 наведено приклад використання декількох `except`-блоків із явним зазначенням типу помилки. Фактично, там розв'язується та сама задача, що й у попередньому випадку, але тільки тепер використано дещо інший спосіб обробки виняткових ситуацій.



Лістинг 2.12. Обробка помилок різних типів

```
print("Розв'язуємо рівняння ax = b")
# Початок try-блоку (основний код)
try:
    # Визначаємо перший параметр. Можлива помилка
    # перетворення тексту в число
```

```

a=float(input("Уведіть а: "))
# Визначаємо другий параметр. Можлива помилка
# перетворення тексту в число
b=float(input("Уведіть b: "))
# Розв'язок рівняння. Можлива помилка
# ділення на нуль
x=b/a
# Відображається значення для кореня рівняння.
# Команда виконується, якщо до цього не виникли
# помилки
print("Розв'язок рівняння: x =",x)
# Початок except-блоку (допоміжний код)
except ValueError:
    # Команда виконується, якщо користувач
    # увів некоректне значення
    print("Потрібно було ввести число!")
except ZeroDivisionError:
    # Команда виконується під час спроби
    # ділення на нуль
    print("Увага! На нуль ділити не можна!")
# Команда виконується після блоку try-except
print("Дякуємо, роботу програми закінчено.")

```

За великим рахунком, змінилася, порівняно з попереднім прикладом, лише та частина коду, що пов'язана з except-блоками. Точніше, таких блоків тепер два. Після першого ключового слова except указано називу `ValueError`. До класу `ValueError` належать помилки, що виникають за неузгодження типів (наприклад, функції потрібен аргумент числового типу, а передається текст). У другому except-блоці обробляються помилки класу `ZeroDivisionError`. До цього класу належать помилки, що виникають під час спроби виконати ділення на нуль.

Як бачимо, для кожного except-блоку пропонується свій програмний код (в обох випадках виводиться повідомлення з інформацією про те, що ж трапилося). Якщо під час виконання коду в блоці `try` виникла помилка, її тип спочатку зіставляється з класом помилки `ValueError` у першому except-блоці. Якщо збіг є, то виконується код цього блоку. Якщо збігу нема, перевіряється збіг типу помилки з класом помилки

`ZeroDivisionError` у другому `except`-блоці. Якщо є збіг — виконується код цього `except`-блоку. Якщо ж і тут збігу немає, виконання програми закінчується і з'являється автоматично створене інтерпретатором повідомлення про помилку.

У випадку, коли всі дані введено коректно, результат виконання програми буде таким же, як і раніше. Нижче показано, як виглядатиме результат виконання програми, якщо для змінної `b` буде введено некоректне значення (жирним шрифтом виділені введені користувачем значення):



Результат виконання програми (з лістингу 2.12)

Розв'язуємо рівняння `ax = b`

Уведіть `a`: **3**

Уведіть `b`: **number**

Потрібно було ввести число!

Дякуємо, роботу програми закінчено.

Реакція програми на спробу ділення на нуль буде такою:



Результат виконання програми (з лістингу 2.12)

Розв'язуємо рівняння `ax = b`

Уведіть `a`: **0**

Уведіть `b`: **5**

Увага! На нуль ділити не можна!

Дякуємо, роботу програми закінчено.

Варто також звернути увагу на таку обставину: оброблюються в цьому випадку лише помилки класів `ValueError` і `ZeroDivisionError`. Якщо б теоретично виникла помилка якогось іншого типу, то її б не було перехоплено.



Щодо класів винятків, то крім перерахованих вище, найбільший інтерес із практичної точки зору можуть становити такі:

- виняток `IndexError`: виникає, коли індекс (наприклад, для елемента **списку**) вказано неправильно (виходить за межі допустимого діапазону). Списки обговорюються в одному із наступних розділів.
- виняток `KeyError`: виникає, коли неправильно вказано ключ **словника**. Словники обговорюються в одному із наступних розділів.
- виняток `NameError`: виникає, якщо не вдається знайти локальне або глобальне ім'я (змінну) з певною назвою.
- виняток `SyntaxError`: пов'язаний з наявністю синтаксичних помилок.
- виняток `TypeError`: пов'язаний з несумісністю типів, коли для обробки (наприклад, у функції) потрібне значення певного типу, а передається значення іншого типу.

Деякі з цих винятків ми будемо використовувати (явно або неявно) у прикладах. Докладніше класи винятків і методи роботи з ними буде обговорено після того, як ми познайомимося з класами й об'єктами.

У розглянутому вище прикладі ми створювали для різних типів помилок різні `except`-блоки. Іноді доводиться вирішувати дещо іншу задачу: для декількох типів помилок створювати один `except`-блок. У цьому випадку після ключового слова `except` у круглих дужках через кому перераховуються ті типи помилок, для яких виконується обробка в даному блоці.

В інструкції `try-except` окрім блоків `try` і `except` можуть також використовуватися блоки `else` і `finally`. Загальний шаблон інструкції в цьому випадку такий (жирним шрифтом виділено ключові елементи):

```
try:
    # основний код
except Тип_помилки_1:
    # допоміжний код
except Тип_помилки_2:
    # допоміжний код
...
except Тип_помилки_N:
    # допоміжний код
else:
    # код для випадку, якщо помилки не було
```

Phyton

finally:

код, який виконується завжди

Блок із ключовим словом `else` розташовується після останнього `except`-блоку і містить програмний код, який виконується тільки в тому випадку, якщо під час виконання основного коду в `try`-блочі помилок не було.

Програмний код, розміщений в блоці `finally`, виконується в будь-якому разі, незалежно від того, виникла помилка під час виконання коду `try`-блоку чи ні.



Помилки можна не лише перехоплювати, а й генерувати штучно. Робиться це за допомогою інструкцій `raise` або `assert`. Як це робиться й навіщо, ми обговоримо в останньому розділі — після того, як познайомимося з класами й об'єктами. Там же буде описано, як можна створювати власні класи винятків.

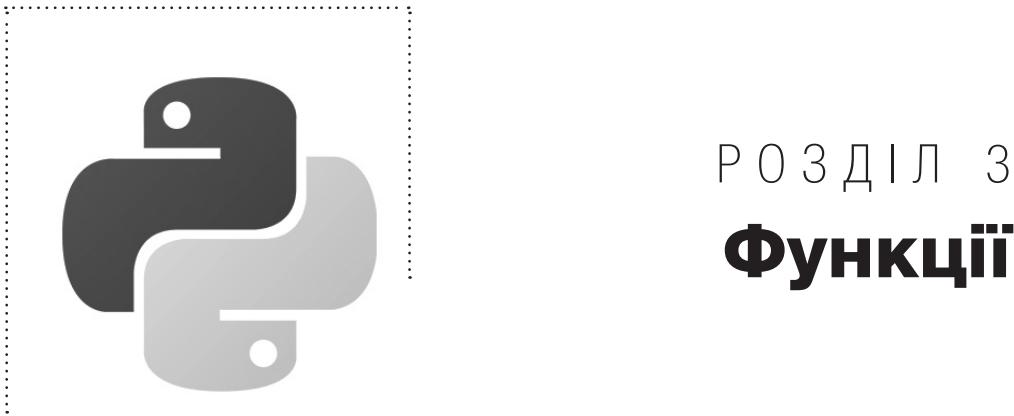
Резюме

Подумати тільки, що через якісні речі можна так зменшитися, що аж перетвориться в ніщо.

Л. Керролл. «Аліса в Країні Див»

- Умовний оператор дозволяє виконувати різні блоки коду залежно від істинності або хибності певної умови.
- В умовному операторі після ключового слова `if` вказується умова, яка перевіряється під час виконання оператора. Якщо умова істинна, виконується блок команд після умови. Якщо умова хибна, виконується блок команд після ключового слова `else`.
- В умовному операторі `else`-блок не є обов'язковим. Також в умовному операторі можуть використовуватися `elif`-блоки, що дозволяє перевіряти в умовному операторі послідовно декілька умов і виконувати для кожної з них окремий блок команд.
- Оператор циклу `while` дозволяє багаторазово виконувати визначений наперед набір команд.
- Після ключового слова `while` в операторі циклу вказується умова. Оператор виконується, поки умова істинна. Умова перевіряється на початку виконання оператора і потім кожен раз після виконання групи команд оператора циклу.

- Оператор циклу `for` зручний у тому випадку, коли необхідно перебрати елементи певної послідовності. Після ключового слова `for` указується змінна для виконання перебору елементів послідовності, яка, у свою чергу, розміщується після ключового слова `in`. Змінна послідовно набуває значення елементів послідовності, і для кожного такого значення виконується блок команд оператора циклу.
- Як послідовність, яка вказується в операторі циклу `for`, можна, крім іншого, використовувати текст, списки або віртуальну числову послідовність, створену за допомогою функції `range()`.
- Інструкції `break` і `continue` використовуються в операторах циклу `while` і `for` відповідно для припинення виконання оператора циклу або для припинення виконання поточного циклу.
- В операторах циклу (`while` і `for`) може використовуватися `else`-блок, який виконується після закінчення роботи оператора і за умови, що закінчення роботи оператора циклу не пов'язане з виконанням інструкції `break`.
- Механізм обробки виняткових ситуацій базується на використанні конструкції `try-except`. Якщо під час виконання програмного коду, поміченого ключовим словом `try`, виникає помилка, її може бути перехоплено і оброблено в одному з блоків, позначених інструкцією `except`. Для кожного `except`-блоку після ключового слова `except` можна вказати тип помилки (винятку), яка оброблюється цим блоком.



Розділ 3

Функції

Частина фактів — це зрозуміти, що у нас є проблема. І частина фактів — це що ми з нею будемо робити.

Дж. Буш (молодший)

Цей розділ присвячено створенню функцій у мові Python. Деякі функції ми вже зустрічали раніше, але це були вбудовані функції. Тут ми дізнаємося, як створювати власні функції. Також ми опануємо основні прийоми використання функцій у програмних кодах.

Коли ми говоримо про функцію, то маємо на увазі фрагмент програмного коду, в якого є назва, і який за цією назвою ми можемо викликати в будь-якому (або практично будь-якому) місці програми. Функції — дуже корисний інструмент програмування, оскільки дозволяє значно скоротити обсяг програмного коду, зробити його більш зрозумілим й результативним. Коротше кажучи, без функцій писати ефективні програми практично нереально. Тому з функціями треба бути на «ти», особливо якщо йдеться про програмування в Python.

Створення функції

В інтересах нашої країни — відшукати тих, хто готовий заподіяти нам зло і послати їх якомога далі.

Дж. Буш (молодший)

Перед тим, як функцію використовувати, її необхідно описати. Що має на увазі опис функції? Як мінімум, це той програмний код, який слід виконувати під час виклику функції, і, зрозуміло, назва функції. Також функції під час виконання можуть знадобитися деякі параметри, які їй передаються під час виклику і без яких виконання функції неможливе. Ці параметри ми будемо називати *аргументами функції*.



Нерідко розрізняють поняття *параметра функції* й *аргументу функції*. Про параметри звичайно говорять під час опису функції. Про аргументи йдеться, коли вже описана функція викликається в програмному коді. Щоб не ускладнювати собі життя, у всіх цих випадках ми будемо використовувати термін *аргумент*.

Отже, для опису функції в Python необхідно:

- задати ім'я функції;
- описати аргументи функції;
- описати програмний код функції.

Опис функції починається з ключового слова `def`. Після нього вказують ім'я функції (це повинен бути унікальний ідентифікатор, бажано зі змістовим навантаженням, який складається з латинських літер, можна також цифр і символу підкреслення — але з цифри ім'я функції починатися

не може). Потім перераховують аргументи функції. Для аргументів просто вказують назви (імена). Тип аргументів не зазначають. Аргументи перераховують після імені функції в круглих дужках. Якщо аргументів декілька, їх розділяють комами. Навіть якщо аргументів у функції немає (таке теж буває), круглі дужки все одно ставлять. Закінчується вся ця конструкція двокрапкою, і далі з нового рядка йде блок команд, які власне і визначають програмний код функції. Весь шаблон оголошення функції виглядає так (жирним шрифтом виділено ключові елементи):

```
def им'я_функції(аргументи):  
    команди
```

Тобто все достатньо просто й прозоро.



Для функції не зазначають тип результату (хоча функція, зрозуміло, може повертати результат). Це може стати сюрпризом для тих, хто вивчав і знайомий з іншими мовами програмування. Але це сюрприз тільки на перший погляд. Якщо згадати, що в мові Python для змінних тип не вказується і визначається за значенням, на яке посилається змінна, то все виглядає цілком логічним: і відсутність ідентифікатора типу для результату функції, її аргументи, для яких тип теж не вказують.

Функція може повернати результат, а може не повернати (в останньому випадку був би сенс говорити про *процедуру* — але в Python усе називається одним словом *функція*). Якщо функція не повертає результат, то її можна розглядати як деяку послідовність команд, які виконуються в тому місці й у той час, що й виклик функції. Якщо функція повертає результат (а це може бути значення практично будь-якого типу), то інструкцію виклику функції можна використовувати у виразах так, як наче б це була якась змінна. У тілі функції, щоб визначити те значення, яке є результатом функції, зазвичай використовують інструкцію `return`. Виконання цієї інструкції, по-перше, приводить до закінчення виконання програмного коду функції, а по-друге, те значення, яке вказано після інструкції `return`, функція повертає як результат.



Викликається функція просто — вказують ім'я функції й аргументи, які їй передаються. Ну љ, зрозуміло, викликати функцію можна тільки після того, як функцію оголошено, але ніяк не раніше.

Далі ми розглянемо простий приклад, у якому оголошуються декілька простих функцій. Після оголошення ці функції викликаються в програмному коді для виконання нескладних дій. Відповідний програмний код наведено в лістингу 3.1.

Лістинг 3.1. Оголошення функцій

```
# Функція без аргументів
def your_name():
    # Відображається повідомлення
    print("Добрий день!")
    # Запам'ятовується введений користувачем текст
    name=input("Як Вас звати? ")
    # Результат функції
    return name

# Функція з одним аргументом
def say_hello(txt):
    # Відображається повідомлення
    print("Вітаю,",txt+"!")

# Викликаємо функцію й результат записуємо в змінну
my_name=your_name()
# Викликаємо функцію з аргументом
say_hello(my_name)
```

У результаті виконання цього програмного коду отримуємо такий результат (жирним шрифтом виділено введене користувачем значення):

Результат виконання програми (з лістингу 3.1)

```
Добрий день!
Як Вас звати? Олексій Васильєв
Вітаю, Олексій Васильєв!
```

Код достатньо простий, але ми його все ж прокоментуємо. У програмі оголошено дві функції. Функція `your_name()` не має аргументів. Під час виконання цієї функції спочатку команда `print("Добрий день!")` відображає привітання. Потім користувачу пропонують увести своє ім'я. Уведене користувачем текстове значення запам'ятовується в змінній `name`. Уся команда виглядає як `name=input("Як Вас звати? ")`. Після цього за допомогою інструкції `return name` значення змінної `name` повертається як результат функції `your_name()`. Таким чином, у цієї функції немає аргументів, проте вона повертає результат. І її результат — це те, що вводить користувач (передбачається, що ім'я користувача).

Ще одна функція `say_hello()` потрібна для відображення привітання. У функції один аргумент (позначено як `txt`). Текст повідомлення, яке відображається під час виклику функції, формується з урахуванням значення аргументу `txt`, переданого функції. Результат функція не повертає. У тілі функції всього одна команда `print("Вітаю, " + txt + "!")`, яка в консольне вікно виводить повідомлення.



Обчислюючи вираз `txt + "!"` ми неявно передбачаємо, що аргумент `txt` посилається на текстове значення. Якщо такої впевненості немає, то краще перестрахуватися й скористатися виразом `str(txt) + "!"`, у якому виконується явне зведення аргументу `txt` до текстового типу.

Описані функції використовуємо таким чином. Спочатку за допомогою команди `my_name=your_name()` викликаємо функцію `your_name()` і результат виклику записуємо в змінну `my_name`. Потім за допомогою команди `say_hello(my_name)` викликаємо функцію `say_hello()`, передавши їй аргументом змінну `my_name`. Результат цих дій такий, як показано вище.

Функції для математичних обчислень

Поза сумнівом, це бюджет — там багато цифр.

Дж. Буш (молодший)

Дуже зручно створювати функції для виконання математичних розрахунків. Особливо природним такий підхід вбачається для реалізації в програмі математичних функцій. Причина в тому, що концепція математичної функції досить близька до концепції функції в програмуванні. Багато найбільш популярних і часто використовуваних математичних функцій зібрано в математичному модулі `math`. Але часто доводиться описувати математичні функції самостійно. Як ілюстрацію розглянемо програмний код у листингу 3.2, у якому реалізовано математичну функцію для обчислення експоненти $\exp(x)$.



Під час обчислень ми використовуємо такий вираз (який є рядом Тейлора для відповідної функції): $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$. Знак оклику `n!` означає обчислення факторіала: за визначенням, $n!$ означає добуток натуральних чисел від 1 до n , тобто $n! = 1 \cdot 2 \cdot 3 \cdots \cdot (n-1) \cdot n$.

Листинг 3.2. Математичні функції

```
# Функція для обчислення експоненти
def my_exp(x, n):
    s=0 # Початкове значення суми ряду
    q=1 # Початкове значення добавки
    # Оператор циклу для обчислення ряду
```

Phyton

```
for k in range(n+1):
    s+=q      # Добавка до суми
    q*=x/ (k+1) # Нова добавка
# Результат функції
return s

# Перевіряємо результат виклику функції
x=1 # Аргумент для експоненти
# Оператор циклу для багатократного
# виклику функції обчислення експоненти
for n in range(11):
    # Відображаємо результат виклику
    # функції обчислення експоненти
    print("n =",n,"->",my_exp(x,n))
```

Функція для обчислення експоненти називається `my_exp()` і має два аргументи: через `x` позначено безпосередньо аргумент для обчислення експоненти, а другий аргумент `n` визначає кількість доданків у сумі для експоненти.



Точний вираз для експоненти $\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$ – це нескінчений ряд. При обчисленні значення для експоненти на практиці використовується наближений вираз $\exp(x) \approx \sum_{k=0}^n \frac{x^k}{k!}$, у якому нескінчений ряд замінено на скінченну суму. Чим більше доданків у сумі, тим точніший вираз для експоненти. У прикладі, що розглядається, аргументами функції, яку ми описуємо для обчислення експоненти, є параметри `x` і `n`.

У тілі функції ми використовуємо декілька змінних. У змінну `s` ми будемо записувати суму ряду для експоненти. Початкове значення цієї змінної дорівнює нулю. Ще одна змінна `q` із початковим одиничним значенням потрібна нам для того, щоб записувати в цю змінну значення добавки, яка на кожній новій ітерації додається до суми ряду.

Сума ряду обчислюється за допомогою оператора циклу. Індексна змінна `k` «пробігає» значення від 0 до `n` включно (вираз `range(n+1)` повертає віртуальну послідовність у діапазоні від 0 до `n` включно). В операторі циклу

всього дві команди: командою `s+=q` сума ряду `s` збільшується на значення добавки `q`, а командою `q*=x/(k+1)` обчислюється значення добавки до ряду для наступного циклу.



При обчисленні суми $\exp(x) \approx \sum_{k=0}^n \frac{x^k}{k!}$ на k -й ітерації добавка до суми ряду до-

рівнює $\frac{x^k}{k!}$, а для наступної ітерації ця добавка повинна бути $\frac{x^{k+1}}{(k+1)!}$. Позначи-

мо $q_k = \frac{x^k}{k!}$ і $q_{k+1} = \frac{x^{k+1}}{(k+1)!}$. Нескладно помітити, що $\frac{q_{k+1}}{q_k} = \frac{x}{k+1}$. Тому,

щоб обчислити добавку для наступного циклу за значенням добавки на поточному циклі (змінна `q`), потрібно поточне значення добавки помножити на величи-

ну $\frac{x}{k+1}$.

Після закінчення роботи оператора циклу сума ряду для експоненти обчислена, і змінна `s` посилається на це значення. Тому командою `return s` значення змінної `s` повертається як результат функції.

У програмі ми перевіряємо результат виклику функції обчислення експоненти для аргументу `x=1`, але за різних значень другого аргументу функції. Для перебору значень (у діапазоні від 0 до 10) другого аргументу функції `my_exp()` запускаємо оператор циклу. У кожному циклі виконується команда `print("n =", n, "->", my_exp(x, n))`. Як наслідок, у вікні виводу відображається послідовність значень другого аргументу і значення функції, обчислене за відповідною кількістю доданків у сумі ряду. Результат виконання програми виглядає так:



Результат виконання програми (з лістингу 3.2)

```
n = 0 -> 1
n = 1 -> 2.0
n = 2 -> 2.5
n = 3 -> 2.666666666666665
n = 4 -> 2.70833333333333
n = 5 -> 2.716666666666663
n = 6 -> 2.718055555555554
n = 7 -> 2.7182539682539684
```

Phyton
.....

```
n = 8 -> 2.71827876984127  
n = 9 -> 2.7182815255731922  
n = 10 -> 2.7182818011463845  
.....
```

Як і слід було очікувати, зі збільшенням кількості доданків точність обчислень зростає (нагадаємо, «точне» значення дорівнює $e \equiv \exp(1) \approx 2.718281828459045$).



Для обчислення експоненти в модулі math є функція `exp()`.

Значення аргументів за замовчуванням

Українаживо зрозуміти, що існує набагато більше торгівельних відносин, ніж торгівлі.

Дж. Буш (молодший)

Для аргументів функцій нерідко задають значення за замовчуванням. Тобто для аргументів функції допускається вказати значення, яке буде використано, якщо аргумент функції явно не зазначений. Можна й інакше: якщо під час виклику функції аргумент не вказанний, то замість нього використовується значення за замовчуванням.

Щоб задати значення аргументу за замовчуванням, в описі функції після імені цього аргументу через знак рівності вказують значення за замовчуванням. У функції, як ми знаємо, можлива наявність декількох аргументів. Частина цих аргументів може мати значення за замовчуванням, а інші аргументи можуть не мати значень. Важливо пам'ятати, що в описі функції під час перерахування аргументів спочатку розміщують аргументи без значень за замовчуванням, а аргументи зі значеннями за замовчуванням знаходяться в кінці.



Такий спосіб передачі функції аргументів (спочатку без значень за замовчуванням, а потім зі значеннями за замовчуванням) — вимушений захід. В іншому разі під час виклику функції, якщо список переданих аргументів неповний, проблематично було б визначити, які саме аргументи передано функції.

Приклади функцій зі значеннями аргументів за замовчуванням наведено в програмному коді лістингу 3.3.

Лістинг 3.3. Значення аргументів за замовчуванням

```
# 1-а функція з одним аргументом.  
# В аргументу є значення за замовчуванням  
def print_text(txt="Значення аргументу за замовчуванням."):   
    print(txt)  
# 2-а функція з двома аргументами.  
# У другого аргументу є значення за замовчуванням  
def show_args(a,b="Другий аргумент не зазначено."):   
    print(a,b)  
# 3-я функція з двома аргументами.  
# В аргументів є значення за замовчуванням  
def my_func(x="1-й аргумент x.",y="2-й аргумент y.,"):   
    print(x,y)  
# Перевіряємо роботу 1-ї функції.  
# Функції передано один аргумент  
print_text("Аргумент указанний явно.")  
# Функції аргументи не передаються  
print_text()  
# Перевіряємо роботу 2-ї функції.  
# Функції передано два аргументи  
show_args("Перший аргумент.", "Другий аргумент.")  
# Функції передано один аргумент  
show_args("Перший аргумент.")  
# Перевіряємо роботу 3-ї функції.  
# Функції аргументи не передаються  
my_func()  
# Функції передано один аргумент  
my_func("Один із аргументів.")  
# Функції передано один аргумент.  
# Переданий аргумент ідентифіковано явно  
my_func(y="Один із аргументів.")
```

При виконанні програмного коду отримуємо такий результат:

Результат виконання програми (з лістингу 3.3)

Аргумент указаній явно.

Значення аргументу за замовчуванням.

Перший аргумент. Другий аргумент.

Перший аргумент. Другий аргумент не зазначено.

1-й аргумент `x`. 2-й аргумент `y`.

Один із аргументів. 2-й аргумент `y`.

1-й аргумент `x`. Один із аргументів.

Ми оголошуємо три функції. Усі три функції діють за однією схемою: значення аргументів виводяться на екран. Маємо на увазі, що всі три функції оперують текстовими аргументами. Деякі з цих аргументів мають значення за замовчуванням. Так, у функції `print_text()` один аргумент, і в цього аргументу є значення за замовчуванням. Ми викликаємо дану функцію без аргументу (у цьому разі використовується значення аргументу за замовчуванням), а також викликаємо її з одним аргументом (у цьому разі використовується передане аргументом значення).

У функції `show_args()` два аргументи, і в другого аргументу є значення за замовчуванням. Тому якщо викликати функцію з одним аргументом, то для другого аргументу використовується значення за замовчуванням. Якщо передаються два аргументи, то для обох аргументів використовуються ті значення, що передані.

У функції `my_func()`, як і в попередньому випадку, два аргументи, але тепер в обох аргументів є значення за замовчуванням. Якщо ми викличемо цю функцію з двома аргументами, то інтриги не буде — значення, передані як аргументи, буде використано під час виконання коду функції. Якщо функції під час виклику передано один аргумент, то виникає питання: який це аргумент — перший чи другий? Адже значення за замовчуванням є і в першого, і в другого аргументу. Тому теоретично значення, передане функції, може бути як першим, так і другим аргументом. За замовчуванням, уважається, що функції передається перший аргумент, а для другого використовується значення за замовчуванням. Але у нас є ще одна можливість: ми можемо явно вказати, для якого аргументу задане значення. У цьому разі під час виклику функції вказується не тільки значення аргументу, а й його *ім'я* (те, яке зазначалося під час опису функції). Використовується формат `ім'я_аргументу=значення`.

Прикладом може бути команда `my_func(y="Один із аргументів.")`. У цьому разі для першого аргументу використовується значення за замовчуванням, а передане функції значення — це значення другого аргументу.

Таким чином, передача аргументів під час виклику функції можлива:

- без зазначення імені аргументів, при цьому значення аргументів передаються строго в тій послідовності, як ці аргументи було описано у функції (такий спосіб передачі аргументів називають *позиційним*);
- із явним зазначенням імені аргументів, при цьому порядок перерахування аргументів не має значення (такий спосіб називається передачею аргументів *за ключем* або *за ім'ям*, а самі аргументи іноді називають *іменованими*).

Якщо під час виклику функції частина аргументів передається за ключем, а частина — позиційним способом, то в команді виклику спочатку зазначаються позиційні аргументи, а потім ті, що передаються за ключем. Варто також звернути особливу увагу на те, що спосіб опису аргументів (при визначенні функції) не залежить від того, як передбачається передавати аргументи під час виклику функції (за ключем чи позиційно).



Якщо в аргументів деякої функції є значення за замовчуванням, то ця функція може викликатися з різною кількістю аргументів (подробиці залежать від специфіки опису функції). У мові Python можна оголошувати функції зі змінною кількістю аргументів. Це питання більш детально ми обговоримо пізніше. Тут же зробимо загальне вступне зауваження.

Припустімо, нам потрібно описати таку функцію, щоб її можна було викликати з різною кількістю аргументів. Важливо, що кількість аргументів не обмежена — ми наперед не знаємо, скільки, у принципі, буде передаватися аргументів функції. У цьому разі ми описуємо функцію з одним аргументом, але перед цим аргументом ставимо зірочку *. Такий аргумент ототожнюється зі списком, елементи якого формуються реальними аргументами, переданими функції під час виклику. Іншими словами, наш «зірковий» аргумент у тілі функції оброблюється як список. Але під час виклику функції аргументи передаються, як звичайно. Наприклад, функцію можна описати так:

```
def get_sum(*nums):  
    s=0  
    for a in nums:  
        s+=a  
    return s
```

Така функція як результат буде повертати суму чисел, переданих їй аргументами. Скажімо, значенням виразу `get_sum(1, 3, 5, 2)` буде число 11, а значенням виразу `get_sum(-2, 4)` – число 2. Списки розглянемо в наступному розділі. Функції зі змінною кількістю аргументів описано в останньому розділі.

Функція як аргумент

Наші вороги винахідливі й кмітливі.
Але ми теж не гірші.

Дж. Буш (молодший)

Щоб зрозуміти наступний прийом, важливо уточнити деякі особливості, що стосуються оголошення функцій у Python. А саме, в результаті оголошення функції насправді створюється об'єкт типу `function`. Посилання на цей об'єкт записується у змінну, яку вказано в описі функції після інструкції `def`. Іншими словами, ім'я функції (без круглих дужок) можна використовувати як змінну (яка посилається на функцію). Питання тільки в тому, що з цією змінною можна зробити. Варіантів тут доволі багато, але нас цікавить, насамперед, можливість присвоїти ім'я функції іншій змінній, а потім викликати таку функцію через цю змінну так, як наче це було б ім'я функції. Під час присвоювання змінній як значення імені функції ця змінна стає посиланням на функцію (нарівні з іменем функції). Невеликий приклад, що ілюструє цю можливість, наведено в лістингу 3.4.

Лістинг 3.4. Посилання на функцію

```
# Вихідна функція
def my_func(txt):
    print("Функція my_func:", txt)
# Змінній присвоюємо ім'я функції
new_func=my_func
# Викликаємо функцію через змінну
new_func("виклик через new_func.")
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 3.4)

Функція `my_func`: виклик через `new_func`.

Ідею тут реалізовано дуже просту. Спочатку ми описуємо функцію `my_func()` (у функції один аргумент), а потім командою `new_func=my_func` змінній `new_func` як значення присвоюється ім'я функції `my_func`. У результаті її ідентифікатор (ім'я функції) `my_func`, і змінна `new_func` посилається на один і той же об'єкт — об'єкт функції. Це означає буквально наступне: про змінну `new_func` ми можемо думати, як про ім'я функції, причому функції тієї ж самої, що й функція `my_func()`. Тому коли виконується команда `new_func("виклик через new_func.")`, це все одно, якби ми викликали функцію `my_func()` із тим самим аргументом.

Описана властивість функцій (можливість присвоювати ім'я функції змінній) має ряд корисних застосувань, у тому числі, перед нами відкривається можливість передавати ім'я функції аргументом іншій функції. Скажімо відверто, можливість нетривіальна і дозволяє легко (і десь навіть красиво) вирішувати складні завдання. Класичною ілюстрацією можуть бути деякі математичні задачі, в яких функціональна залежність відіграє роль зовнішнього фактора: наприклад, під час обчислення інтегралів, розв'язання диференціальних або алгебраїчних рівнянь.

Далі ми розглянемо декілька ілюстрацій до того, як одну функцію передають аргументом іншій функції. Розпочнемо з простого прикладу, в якому описано функцію для розв'язання алгебраїчних рівнянь методом послідовних наближень.



Ідеється про розв'язання рівняння вигляду $x = f(x)$ відносно змінної x , за умови, що функція $f(x)$ відома (задана). Суть методу послідовних наближень полягає в тому, що задано початкове наближення x_0 для кореня рівняння, на основі якого обчислюється перше наближення для кореня $x_1 = f(x_0)$. На основі першого наближення x_1 обчислюють друге наближення $x_2 = f(x_1)$ і так далі. Загальна рекурентна формула для обчислення x_{n+1} наближення

на основі наближення x_n має вигляд $x_{n+1} = f(x_n)$. Саме цю схему ми збираємося реалізувати в програмі.

Зрозуміло, що далеко не кожне рівняння можна розв'язати таким методом. Існують певні критерії застосовності методу послідовних наближень. Зокрема, функція $f(x)$ повинна бути такою, щоб на всій області пошуку кореня виконувалося співвідношення $|f'(x)| < 1$, тобто модуль похідної від функції повинен бути менший від одиниці.

Задача полягає в тому, щоб не просто розв'язати якесь конкретне рівняння, а написати функцію, яка б дозволяла розв'язувати зазначеним методом різні рівняння (зрозуміло, за умови, що метод узагалі для них застосовний). Програмний код із розв'язанням цієї задачі наведено в листингу 3.5.

Листинг 3.5. Метод послідовних наближень

```
# Опис функції для розв'язання рівняння
def solve_eqn(f,x0,n):
    # Початкове наближення для кореня
    x=x0
    # Оператор циклу для обчислення
    # наближень для розв'язку
    for k in range(1,n+1):
        x=f(x) # Ітераційна формула
        # Результат функції
    return x

# Функція, що визначає 1-е рівняння
def eqn_1(x):
    # Значення функції
    return (x**2+5)/6

# Функція, що визначає 2-е рівняння
def eqn_2(x):
    # Значення функції
    return (6*x-5)**0.5

# Розв'язуємо 1-е рівняння
x=solve_eqn(eqn_1,0,10)
# Відображаємо результат
```

```

print("1-е рівняння: x =",x)
# Розв'язуємо 2-е рівняння
x=solve_eqn(eqn_2,4,10)
# Відображаємо результат
print("2-е рівняння: x =",x)

```

У програмі ми описуємо функцію `solve_eqn()` з трьома аргументами. Наші ідеї щодо аргументів такі:

- перший аргумент f є посиланням на функцію, що визначає розв'язуване рівняння (мається на увазі функція $f(x)$ у рівнянні $x = f(x)$);
- другий аргумент x_0 визначає початкове наближення для кореня рівняння;
- третій аргумент n функції — кількість ітерацій, за якими обчислюється корінь.

У тілі функції `solve_eqn()` командою `x=x0` змінній x як початкове присвоюється нульове наближення для кореня рівняння. Після цього запускається оператор циклу, в якому здійснюється n ітерацій (циклів). За кожен цикл виконується команда `x=f(x)`, у якій ми використовуємо виклик функції, переданої через ідентифікатор f аргументом функції `solve_eqn()`. Кожне виконання команди `x=f(x)` приводить до обчислення нового (наступного) наближення для кореня рівняння. Після завершення оператора циклу змінна x командою `return x` повертається як результат функції `solve_eqn()`.

Також ми визначаємо дві функції від одного аргументу, які задають рівняння для розв'язання.



Ми розв'язуємо квадратне рівняння $x^2 - 6x + 5 = 0$, яке має два корені: $x = 1$ і $x = 5$. Для застосування методу послідовних наближень дане рівняння потрібно переписати. Причому для пошуку різних коренів потрібно використовувати різні зображення. Пов'язано це з необхідною умовою збіжності методу. Так, для пошуку кореня $x = 1$ рівняння подамо у вигляді $x = \frac{x^2 + 5}{6}$. У цьому разі рівняння задається функцією $f(x) = \frac{x^2 + 5}{6}$. Похід-

на від цієї функції $f'(x) = \frac{x}{3}$ у точці шуканого кореня $x = 1$ за модулем мен-

ша від одиниці. Функція $f(x) = \frac{x^2 + 5}{6}$ у програмному коді реалізується за допомогою функції `eqn_1()`. Ім'я цієї функції будемо вказувати першим аргументом функції `solve_eqn()`. Початкове наближення в такому разі повинно потрапляти в інтервал $-3 < x < 3$. Другий корінь рівняння $x = 5$ не потрапляє до цього інтервалу, тому для його обчислення необхідно змінити спосіб запису рівняння. Тепер представимо рівняння у вигляді $x = \sqrt{6x - 5}$. Це рівняння задано функцією $f(x) = \sqrt{6x - 5}$. Похідна від даної функції

$$f'(x) = \frac{3}{\sqrt{6x - 5}}$$
 за модулем менша від одиниці при $x > \frac{7}{3} \approx 2.66667$. За-

лежність $f(x) = \sqrt{6x - 5}$ у програмі реалізовано через функцію `eqn_2()`. Ім'я цієї функції передамо першим аргументом функції `solve_eqn()`. Другим аргументом потрібно вказати числове значення, більше за $7/3$.

Хоча в даному випадку фактично йдеться про одне рівняння, записане по-різному, ми, щоб не плутатися, говоритимемо про два рівняння.

Функція `eqn_1()`, що визначає перше рівняння, містить лише одну команду `return (x**2+5) / 6`, якою на основі значення аргументу x обчислюється результат функції. У цьому разі визначається функціональна залежність $f(x) = \frac{x^2 + 5}{6}$ і розв'язується, відповідно, рівняння $x = \frac{x^2 + 5}{6}$ (ідеться про корінь $x = 1$).

Ще одна функція, яка називається `eqn_2()`, також призначена для визначення рівняння. Значення функції повертається командою `return (6*x-5)**0.5`, де x — аргумент функції. Таким чином, розглядаємо функцію $f(x) = \sqrt{6x - 5}$ для рівняння $x = \sqrt{6x - 5}$ (і шукаємо розв'язок $x = 5$).

Для розв'язання першого рівняння ми використовуємо команду `x=solve_eqn(eqn_1, 0, 10)`. Значенням змінній x присвоюється результат виклику функції `solve_eqn()`, першим аргументом якій передано ім'я `eqn_1` функції, що визначає розв'язуване рівняння, другий аргумент — початкове (у цьому разі нульове) значення для кореня рівняння, а третій аргумент (значення 10) визначає кількість ітерацій, за якими обчислюється наближення для кореня рівняння (у принципі, чим більше ітерацій — тим вища точність розв'язку). Командою

`print("1-е рівняння: x =", x)` знайдений розв'язок для кореня рівняння відображається в консольному вікні.

За тією ж схемою шукають розв'язок для другого рівняння, тільки тепер першим аргументом функції `solve_eqn()` передається ідентифікатор `eqn_2`, а початкове значення для кореня (другий аргумент) дорівнює 4. Результат виконання програми показано нижче:

Результат виконання програми (з лістингу 3.5)

```
1-е рівняння: x = 0.9999925289581152
2-е рівняння: x = 4.992839487820055
```

Як нескладно помітити, знайдені нами наближені розв'язки досить близькі до точних значень.

Далі розглянемо задачу про розв'язання диференціального рівняння першого порядку $y'(x) = f(x, y(x))$ із початковою умовою $y(x_0) = y_0$ (*задача Коши*). Тут через x позначено незалежну змінну, $y(x)$ — невідома функція від незалежної змінної, $y'(x) \equiv \frac{dy}{dx}$ — похідна від функції $y(x)$ за аргументом x , а $f(x, y)$ — відома (задана) функція двох аргументів (функцію $f(x, y)$ будемо називати *функцією рівняння*). Нам необхідно знайти таку функцію $y(x)$, щоб диференціальне рівняння $y'(x) = f(x, y(x))$ перетворилося на тотожність, і, крім цього, у точці $x = x_0$ функція $y(x)$ набувала значення y_0 (числові параметри x_0 і y_0 вважаємо заданими). Для розв'язання рівняння в числовому вигляді застосуємо найпростішу схему Ейлера.



Якщо коротко, то суть схеми Ейлера полягає в тому, що інтервал значень аргументу від x_0 (точка, в якій задано початкову умову) до x (точка, в якій потрібно обчислити значення функції $y(x)$) розбивають на n (досить велике число) одинакових відрізків довжиною $\Delta x = \frac{x - x_0}{n}$. Значення функції $y(x)$ послідовно обчислюють у вузлових точках $x_k = x_0 + \Delta x \cdot k$, де індекс $k = 0, 1, 2, \dots, n$, причому, за домовленістю, $x_n \equiv x$. У вузловій точці x_0 значення функції $y(x_0) = y_0$ визначають з початкової умови. Для всіх інших вуз-

лових точок x_k обчислення значення функції $y_k \equiv y(x_k)$ здійснюють на основі рекурентного спiввiдношення $y_k = y_{k-1} + \Delta x \cdot f(x_{k-1}, y_{k-1})$. Таким чином, якщо ми знаємо значення функції y_0 у точці x_0 (а ми його знаємо з початкової умови), то можемо обчислити значення функції y_1 у точці x_1 за формулою $y_1 = y_0 + \Delta x \cdot f(x_0, y_0)$. Знаючи y_1 , обчислюємо $y_2 = y_1 + \Delta x \cdot f(x_1, y_1)$ і так далі до $y_n = y_{n-1} + \Delta x \cdot f(x_{n-1}, y_{n-1})$. А $y_n \equiv y(x_n)$ — це і є $y(x)$.

Для реалізації схеми Ейлера з обчислення розв'язку диференціального рівняння в точці описуємо спеціальну функцію (називається `solve_deqn()`). У цієї функції чотири аргументи:

- назва функції, що визначає диференціальне рівняння (аргумент із назвою `f`);
- вузлова точка, в якій задається початкова умова (аргумент із назвою `x0`);
- значення шуканої функції в точці початкової умови (аргумент із назвою `y0`);
- значення точки, для якої необхідно обчислити значення функції — розв'язок диференціального рівняння (аргумент із назвою `x`).

Як результат функція буде повертати значення розв'язку диференціального рівняння (для заданої точки). Відповідний програмний код наведено в листингу 3.6.

Листинг 3.6. Розв'язання диференціального рівняння

```
# Імпорт математичного модуля
import math
# Функція для розв'язання диференціального рівняння
def solve_deqn(f,x0,y0,x):
    # Кількість відрізків, на які ділиться інтервал
    # пошуку розв'язку рівняння
    n=1000
    # Відстань між сусіднimi вузловими точками
    dx=(x-x0)/n
    # Початкова точка
    x=x0
    # Початкове значення функції
    y=y0
```

```

# Оператор циклу для обчислення розв'язку
for k in range(1,n+1):
    # Значення функції у вузловій точці
    y=y+dx*f(x,y)
    # Наступна вузлова точка
    x=x+dx
# Результат функції
return y

# Функція, що визначає диференціальне рівняння
def diff_eqn(x,y):
    # Результат функції
    return 2*x-y

# Функція для точного розв'язку рівняння
def y(x):
    # Результат функції
    return 2*(x-1)+5*math.exp(-x)

# Крок приросту за аргументом
h=0.5

# Обчислення результату для декількох
# значень аргументу
for k in range(0,6):
    # Значення аргументу
    x=k*h
    print("Числовий розв'язок:")
    # Числовий розв'язок
    print("x =",x,"-> y(x) =",solve_deqn(diff_eqn,0,3,x))
    print("Точний розв'язок:")
    # Точний розв'язок
    print("x =",x,"-> y(x) =",y(x))

```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 3.6)

Числовий розв'язок:

x = 0.0 -> y(x) = 3.0

Точний розв'язок:

x = 0.0 -> y(x) = 3.0

Phython

Числовий розв'язок:

$x = 0.5 \rightarrow y(x) = 2.0322741142003067$

Точний розв'язок:

$x = 0.5 \rightarrow y(x) = 2.032653298563167$

Числовий розв'язок:

$x = 1.0 \rightarrow y(x) = 1.8384771238548225$

Точний розв'язок:

$x = 1.0 \rightarrow y(x) = 1.8393972058572117$

Числовий розв'язок:

$x = 1.5 \rightarrow y(x) = 2.1143951442170557$

Точний розв'язок:

$x = 1.5 \rightarrow y(x) = 2.115650800742149$

Числовий розв'язок:

$x = 2.0 \rightarrow y(x) = 2.6753226122334164$

Точний розв'язок:

$x = 2.0 \rightarrow y(x) = 2.6766764161830636$

Числовий розв'язок:

$x = 2.5 \rightarrow y(x) = 3.4091422819998414$

Точний розв'язок:

$x = 2.5 \rightarrow y(x) = 3.410424993119494$

Хоча програмний код чималий, програма насправді проста. Проаналізуймо основні її місця й найважливіші моменти.

Відносно нова для нас інструкція `import math` потрібна для імпорту математичного модуля `math`, а він, у свою чергу, знадобиться нам для того, щоб при перевірці знайденого числового розв'язку (порівнянні його з точним, аналітичним розв'язком), можна було використовувати функцію для обчислення значення експоненти `exp()`.

У тілі функції `solve_deqn()`, використаної для розв'язання диференціального рівняння, команда `n=1000` задає кількість відрізків, на які розбито інтервал пошуку розв'язку (мається на увазі інтервал значень аргументу функції-розв'язку диференціального рівняння від x_0 до x). Тоді відстань між сусідніми вузловими точками становитиме величину $dx = (x - x_0) / n$ (довжина інтервалу, поділена на кількість відрізків, на які

розділі інтервал). У змінну x за допомогою команди $x=x0$ записується значення точки, в якій задано початкову умову, а командою $y=y0$ змінній y присвоюється значення шуканої функції в точці початкової умови. Таким чином, змінні x і y отримують свої початкові значення. Після цього запускається ітераційний процес, основу якого складає оператор циклу. Оператор циклу розрахованій на виконання n ітерацій, при чому за кожну ітерацію виконується дві команди. Спочатку командою $y=y+dx*f(x,y)$ обчислюється значення функції-розв'язку рівняння в наступній (порівняно з поточного) вузловій точці. Потім командою $x=x+dx$ обчислюється сама вузлова точка. Після виконання оператора циклу значення змінної y повертається як результат функції `solve_deqn()`.

Також у програмі ми визначаємо функцію, яка задає розв'язуване диференціальне рівняння. Ідеється про функцію `diff_eqn()`, у якої два аргументи (x і y), а як результат функція повертає вираз $2*x-y$.



Виходячи з визначення функції `diff_eqn()`, нескладно зробити висновок, що йдеться про функцію рівняння $f(x,y) = 2x - y$. Отже, розв'язується диференціальне рівняння $y'(x) = 2x - y(x)$. У цього рівняння є аналітичний (точний) розв'язок $y(x) = 2 \cdot (x - 1) + C_1 \cdot \exp(-x)$, де постійна C_1 визначається з початкової умови. Наприклад, якщо початкову умову записано як $y(0) = 3$ (а в програмі під час перевірки результату ми використовуємо саме таку початкову умову), то розв'язок задачі Коші матиме вигляд $y(x) = 2 \cdot (x - 1) + 5 \cdot \exp(-x)$.

Щоб порівняти результат, отриманий за схемою Ейлера, із точним розв'язком $y(x) = 2 \cdot (x - 1) + 5 \cdot \exp(-x)$ рівняння, у програмному коді ми оголошуємо функцію `y()` із одним аргументом x , яка відповідає аналітичному розв'язкові задачі Коші для рівняння $y'(x) = 2x - y(x)$ із початковою умовою $y(0) = 3$.



У тілі функції `y()` для обчислення значення експоненти викликається функція `exp()` із модуля `math`, який ми на самому початку програми підключили за допомогою інструкції `import`. Як аргумент функції `exp()` передаємо

показник експоненти. Під час виклику функції `exp()` явно вказується модуль `math`, тому математичному виразу `exp(-x)` (або те саме, що e^{-x}) у програмному коді відповідає інструкція `math.exp(-x)`.

Перевірку створених функцій виконуємо так: для декількох значень аргументу x функції-розв'язку рівняння ми знаходимо числовий розв'язок і порівнюємо його з точним розв'язком. Для цього запускаємо оператор циклу, в якому індексна змінна k «пробігає» значення від 0 до 5. Змінна x набуває значення $k \cdot h$ ($h=0.5$ — крок приросту для аргументу x), наближений (числовий) розв'язок обчислюємо за допомогою інструкції `solve_deqn(diff_eqn, 0, 3, x)`, а для отримання точного розв'язку використовуємо вираз $y(x)$.

Рекурсія

Про незаконність дій нам слід говорити так, ніби у нас її нема.

Дж. Буш (молодший)

Під час опису функцій іноді зручно використовувати *рекурсію*. Рекурсія передбачає, що в програмному коді функції викликається ця сама функція (але зазвичай з іншим аргументом). Краще це проілюструвати прикладом. У лістингу 3.7 наведено програмний код функції, яка за порядковим номером розраховує число з послідовності Фібоначчі. В описі функції використано рекурсію.



Числа Фібоначчі — послідовність чисел, у якій перше й друге числа дорівнюють одиниці, а кожне наступне є сумою двох попередніх. Отже, йдеться про числа 1, 1, 2, 3, 5, 8, 13, 21 і так далі.

Лістинг 3.7. Числа Фібоначчі

```
# Функція для розрахунку чисел Фібоначчі.  
# В описі функції використано рекурсію  
def Fib(n):  
    # Перше і друге число  
    # в послідовності дорівнюють 1  
    if n==1 or n==2:  
        return 1  
    # Числа в послідовності  
    # дорівнюють сумі двох попередніх  
    else:  
        return Fib(n-1)+Fib(n-2)
```

Phyton

```
# Перевіряємо роботу функції
print("Числа Фібоначчі:")
# Обчислюємо 15 перших чисел Фібоначчі
for i in range(1,16):
    # Числа друкуються в одному рядку через пробіл
    print(Fib(i),end=" ")
```

Під час виконання програми в ряд через пробіл відображаються 15 чисел із послідовності Фібоначчі:



Результат виконання програми (з лістингу 3.7)

Числа Фібоначчі:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Функція для розрахунку чисел Фібоначчі в нашому виконанні називається `Fib()`, і в неї один аргумент (позначено як `n`) — це порядковий номер числа в послідовності. За цим номером необхідно обчислити число. Процес обчислень ми реалізуємо за допомогою умовного оператора, в якому перевіряємо значення аргументу функції. Точніше, перевіряємо умову `n==1 or n==2`. Умова істинна, якщо аргумент `n` дорівнює 1 або 2, тобто якщо йдеться про перше або друге число в послідовності. У цьому випадку число Фібоначчі дорівнює 1. Тому якщо умова істинна, виконується інструкція `return 1`. Тобто якщо у функції `Fib()` значення аргументу дорівнює 1 або 2, функція повертає значення 1. Поки все просто. Ситуація ускладнюється, якщо умова `n==1 or n==2` хибна. Хибною є умова, якщо індекс у числа (його порядковий номер у послідовності) більший, ніж 2 (екзотичні варіанти з від'ємними і нецілими індексами ми не розглядаємо). І тут ми згадуємо правило обчислення чисел у послідовності Фібоначчі: кожне число (крім перших двох) — це сума двох попередніх. Далі, якщо число Фібоначчі з порядковим номером `n` повертається в результаті виклику функції командою `Fib(n)`, то два попередніх числа — це `Fib(n-1)` і `Fib(n-2)`. Тому в тілі функції (з аргументом `n`) в умовному операторі в `else`-блоці виконується команда `return Fib(n-1)+Fib(n-2)`. Власне, на цьому все. У нас є програмний код функції, і цей програмний код працює.



Щоб зрозуміти, чому ж рекурсія «працює», розберімося, що відбувається, коли ми викликаємо функцію `Fib()`. Наприклад, якщо ми викликаємо функцію з аргументом 1 або 2 (вираз `Fib(1)` або `Fib(2)` відповідно), то в тілі функції «у гру» вступає той блок в умовному операторі, який відповідає істинній умові. А ось щоб обчислити вираз `Fib(3)` в `else`-блочі умовного оператора виконується спроба обчислити вираз `Fib(2) + Fib(1)`. При цьому знову викликається функція `Fib()`, але тільки з аргументами 2 і 1. Що відбувається в такому випадку, ми вже знаємо. Після того, як обчислено значення `Fib(1)` і `Fib(2)`, може бути обчислено значення `Fib(3)`. Для обчислення значення виразу `Fib(4)` обчислюється сума `Fib(3) + Fib(2)`, у якій потрібно попередньо обчислити `Fib(3)` і `Fib(2)`. Як обчислюються ці значення, ми розглядали вище. Після їхнього обчислення, обчислюється значення `Fib(4)` і так далі.

Для перевірки роботи функції `Fib()` ми за її допомогою обчислюємо і виводимо в один рядок 15 перших чисел із послідовності Фібоначчі.



За замовчуванням, після виведення в консольне вікно функцією `print()` значень своїх аргументів виконується перехід до нового рядка. Тому якщо викликати декілька разів поспіль функцію `print()`, кожне нове повідомлення (текст, що виводиться) буде з'являтися в окремому рядку. Причина в тому, що в кінці тексту, що виводиться в консоль (вікно виводу), автоматично додається інструкція переходу до нового рядка. Щоб змінити цей режим роботи функції `print()` можна в явному вигляді вказати текстове значення для аргументу `end`. Значення зазвичай передають за ключем, через знак рівності після назви аргументу `end`. Це значення визначає символ (текст), який додається (замість інструкції переходу до нового рядка) у кінці тексту, що виводиться. Наприклад, у команді `print(Fib(i), end=" ")` використано інструкцію `end=" "`, тому після виведення значення першого аргументу додається пробіл. Перехід до нового рядка не відбувається.

Хоча рекурсія зазвичай і робить програмні коди компактними та інтуїтивно зрозумілими, метод рекурсивного визначення функції не можна назвати економним у плані використання системних ресурсів. Разом із тим, нерідко на практиці рекурсія є єдино можливим способом реалізації програмного коду функції.

Розгляньмо ще один приклад рекурсії. Цього разу перепишемо приклад із лістингу 3.5, у якому, нагадаємо, наведено програму для розв'язання алгебраїчних рівнянь методом послідовних наближень. Тільки тепер функцію, за допомогою якої реалізується ітераційний процес уточнення розв'язку рівняння, реалізуємо через рекурсію. Звернімось до програмного коду в лістингу 3.8.

Лістинг 3.8. Рекурсія для методу послідовних наближень

```
# Опис функції для розв'язання рівняння.  
# Використовуємо рекурсію  
def solve(f,x0,n):  
    # Початкове наближення  
    if n==0:  
        return x0  
    # Рекурсивне співвідношення  
    else:  
        return solve(f,f(x0),n-1)  
# Функція, що визначає рівняння  
def eqn(x):  
    # Значення функції  
    return (x**2+5)/6  
# Розв'язуємо рівняння  
x=solve(eqn,0,10)  
# Відображаємо результат  
print("Розв'язок рівняння: x =",x)
```

Ми трохи змінили програмний код: подекуди переписали, подекуди спростили. Функція `solve()` призначена для розв'язання рівняння, що визначається першим аргументом (посилання на функцію рівняння `f`) із початковим наближенням, що визначається другим аргументом (позначено як `x0`) за кількістю ітерацій, яка визначається третім аргументом (позначено як `n`).

Основу програмного коду функції складає умовний оператор, у якому перевіряється умова `n==0`. Що означає істинність цієї умови? Істинність цієї умови означає, що ми обчислюємо нульове наближення для кореня

рівняння. Але нульове наближення визначається другим аргументом x_0 функції `solve()`. Тому немає нічого дивного в тому, що за істинної умови $n==0$ командою `return x0` значення x_0 повертається як результат функції `solve()`. Проте якщо умова $n==0$ хибна, результатом повертається вираз `solve(f, f(x0), n-1)`, у якому рекурсивно викликається функція `solve()`.



Щоб зрозуміти походження виразу `solve(f, f(x0), n-1)`, потрібно врахувати таке. Припустімо, нам потрібно обчислити n -ну ітерацію для кореня рівняння $x = f(x)$ із початковим наближенням $x = x_0$. Якщо функція `solve()` дозволяє розв'язувати такі задачі, $f()$ на програмному рівні є реалізацією функції рівняння, параметр $x0$ визначає початкове наближення для кореня, а n — номер ітерації, то формально результат отримуємо за допомогою команди `solve(f, x0, n)`. З іншого боку, обчислення n -ної ітерації на основі нульового наближення — це те ж саме, що обчислення $(n-1)$ -ого наближення на основі першого наближення x_1 . Але $x_1 = f(x_0)$. Тому результат на програмному рівні може бути наведено також виразом `solve(f, f(x0), n-1)`. Тут ми врахували, що перше наближення для кореня рівняння можна обчислити як $f(x0)$.

Результат роботи створеної нами функції `solve()` перевіряємо для розв'язання рівняння $x = \frac{x^2 + 5}{6}$. Із цією метою в програмі описано функцію `eqn()`. У підсумку отримуємо такий результат:



Результат виконання програми (з лістингу 3.8)

Розв'язок рівняння: $x = 0.9999925289581152$

Бачимо, що визначена через рекурсію функція для розв'язання рівняння виконується коректно. Хто бажає, може самостійно переконатися в тому, що за допомогою цієї функції можна знайти другий корінь рівняння. Для цього лише потрібно представити в іншому вигляді саме рівняння — як ми це робили раніше.

Лямбда-функції

Мої погляди — це ті, якими говорить
свобода.

Дж. Буш (молодший)

Ми вже знаємо, що ім'я функції можна присвоїти як значення змінній, після чого через цю змінну допустимо посилатися на функцію. Але в мові Python є й інша «крайність»: ми можемо створити функцію, але ім'я її не присвоювати зовсім. Такі функції називаються *анонімними функціями* або *лямбда-функціями*. Навіщо подібні функції потрібні — це питання окреме. Можна навести як мінімум декілька прикладів, коли використання лямбда-функцій видається вправданим:

- передача лямбда-функції аргументом іншій функції;
- повернення одної функції результатом іншої функції;
- одноразове використання функції.

В описі лямбда-функції використовують ключове слово `lambda`, після якого перераховують аргументи функції, а через двокрапку — її результат. Тобто шаблон опису лямбда-функції такий (жирним шрифтом виділено ключові елементи):

lambda *аргументи*: *результат*

У цій конструкції є значення: посилання на об'єкт лямбда-функції. Тому, в принципі, `lambda`-конструкцію можна присвоїти як значення змінній, тим самим визначивши, загалом, звичайну функцію. Невеликий приклад, що ілюструє деякі аспекти оголошення й використання лямбда-функцій, наведено в лістингу 3.9.

Лістинг 3.9. Лямбда-функції

```
# Функція для відображення значення
# іншої функції
def find_value(f,x):
    print("x =",x,"-> f(x) =",f(x))
# Змінній присвоюється
# посилання на лямбда-функцію
my_func=lambda x: 1/(1+x**2)
# Перевіряємо результат
find_value(my_func,2.0)
# Як аргумент функції передано
# лямбду-функцію
find_value(lambda x: x*(1-x),0.5)
# Використання лямбда-функції
# у виразі
z=1+(lambda x,y: x*y-x**2)(2,3)**2
# Перевіряємо значення змінних
print("z =",z)
```

У програмному коді оголошуємо функцію `find_value()` із двома аргументами. Це — звичайнісінька функція. Ми збираємося її використовувати як допоміжну для ілюстрації до використання лямбда-функцій. Ми виходимо з того, що перший аргумент `f` є посиланням на функцію, а `x` — передбачуваний аргумент для цієї функції. У тілі функції `find_value()` командою `print("x =",x,"-> f(x) =",f(x))` відображається повідомлення, в якому, крім іншого, використано інструкцію `f(x)` виклику функції, на яку посилається змінна `f`, із аргументом `x`.

Далі наведено декілька прикладів опису й використання лямбда-функцій. Так, у команді `my_func=lambda x: 1/(1+x**2)` є опис лямбда-функції, а посилання на цю функцію присвоюється змінній `my_func`. Власне функцію описує інструкція `lambda x: 1 / (1+x**2)`, у якій після ключового слова `lambda` зазначено формальну назву `x` для аргументу функції, а вираз `1 / (1+x**2)` означає, що за аргументу функції `x` її результатом буде значення `1 / (1+x**2)`.



Таким чином, тут ідеться про функцію $f(x) = \frac{1}{1+x^2}$.

Щоб перевірити "працездатність" створеної нами функції, використовуємо команду `find_value(my_func, 2.0)`. У цій команді першим аргументом функції `find_value()` передано ім'я змінної `my_func`, яка містить посилання на лямбда-функцію. Другий аргумент `2.0` функції `find_value()` — це значення аргументу для функції, на яку посилається змінна `my_func`. У результаті в консольне вікно виводиться повідомлення зі значенням `my_func(2.0)`. Результатом є число `0.2` (легко перевірити, що $f(2) = \frac{1}{1+2^2} = \frac{1}{5} = 0.2$).

Ще один варіант, який ми розглядаємо, — передача лямбда-функції аргументом іншій функції. Відповідна команда виглядає як `find_value(lambda x: x*(1-x), 0.5)`. Тут першим аргументом функції `find_value()` передано не змінну з посиланням на лямбда-функцію, як у попередньому випадку, а саму лямбда-функцію. Перший аргумент виглядає так: `lambda x: x*(1-x)`. Тобто аргументу функції `x` у відповідність ставиться вираз `x*(1-x)` (це означає, що ми маємо справу з функцією $f(x) = x \cdot (1-x)$). Другий аргумент функції `find_value()` — числове значення `0.5`. Це значення відіграє роль аргументу для функції, переданої першим аргументом функції `find_value()`. І хоча як перший аргумент передано не змінну, а анонімну функцію, суть спрости не змінюється: обчислюється значення $f(0.5) = 0.5 \cdot (1 - 0.5) = 0.25$, тобто обчислене значення дорівнює `0.25`.

Третій приклад застосування лямбда-функції — використання такої функції у виразі. Прикладом є команда `z=1+(lambda x,y: x*y-x**2)(2,3)**2`, у результаті виконання якої змінна `z` отримує значення `5`. Значення виразу `1+(lambda x,y: x*y-x**2)(2,3)**2` обчислюється так: до одиниці додається значення виразу `(lambda x,y: x*y-x**2)(2,3)**2`. Даний вираз — це квадрат значення виразу `(lambda x,y: x*y-x**2)(2,3)`. А цей вираз, у свою чергу, є не що інше, як результат дії лямбда-функції `(lambda x,y: x*y-x**2)` на аргументи `(2,3)`. Інструкція `lambda x,y: x*y-x**2` визначає лямбда-функцію двох аргументів (`x` і `y`), а результатом є значення `x*y-x**2`, що розраховується на основі значень аргументів функції. Тобто тут маємо справу з функцією $f(x,y) = xy - x^2$. Якщо обчислювати значення цієї функції для аргументів `x = 2, y = 3`,

отримаємо $f(2,3) = 2 \cdot 3 - 2^2 = 6 - 4 = 2$. Отже, значенням виразу `(lambda x,y: x*y-x**2)(2,3)` є 2, значенням виразу `(lambda x,y: x*y-x**2)(2,3)**2` є 4, а значення виразу `1+(lambda x,y: x*y-x**2)(2,3)**2`, таким чином, дорівнює 5.

У результаті виконання програми отримуємо таке повідомлення у вікні виводу:

Результат виконання програми (з лістингу 3.9)

```
x = 2.0 -> f(x) = 0.2
x = 0.5 -> f(x) = 0.25
z = 5
```

Ще одну програму, яка ілюструє можливі варіанти застосування лямбда-функцій, наведено в лістингу 3.10. У цьому разі лямбда-функцію використовуємо для того, щоб створити функцію, яка повертає результатом іншу функцію.



Ми звички, і нам видається цілком природним, що результат виклику функції — це деяке значення (наприклад, число або текст). Тут ідеється про те, що результатом функції є функція. Щоб зрозуміти, як таке можливо (як функція може повертати результатом функцію), варто згадати, що функція в Python — це об'єкт типу `function` (про об'єкти ми будемо говорити в наступних розділах, тут ми вдаватися в подробиці відносно такого поняття, як об'єкт, особливо не будемо). Для нас важливо те, що опис функції — це деякі дані, на які можна посилатися, і записати це посилання в змінну, яка відіграє роль імені функції. Тому якщо ми говоримо, що результатом функції є функція, то означає це, як правило, таке:

- у тілі функції описано (створено) деяку функцію;
- посилання на створену функцію повертається як результат.

Варто відзначити концептуально важливий момент. Коли ми викликаємо функцію (наприклад, командою виду функція (аргументи)), то сама інструкція виклику функції теж є функцією. Тобто вираз функція (аргументи) можна інтерпретувати як ім'я функції (якщо точніше, то посилання на функцію). Це посилання саме на ту функцію, яка повертається як результат функції, що викликається. Тому якщо після інструкції функція (аргументи)

у круглих дужках вказати аргументи, то отримаємо виклик функції-результату. Іншими словами, під час виклику функції-результату маємо справу з виразом вигляду функція(аргументи) (аргументи). Можна зробити інакше: присвоїти деякій змінній значення виразу функція(аргументи). Після виконання команди змінна=функція(аргументи) зі змінною можна поводитися як із ім'ям функції.



Листинг 3.10. Функція як результат функції

```
# Функція як результат
# повертає функцію
def my_pow(n):
    return lambda x: x**n
# Перевіряємо результат
for n in range(1,4): # Зовнішній цикл
    for x in range(1,11): # Внутрішній цикл
        # Виводимо результат виклику функції
        print(my_pow(n)(x),end=" ")
print() # Переходимо до нового рядка
```

У програмному коді описано функцію `my_pow()`, у якої один аргумент `n`. Як результат функції повертається інструкція `lambda x: x**n`. Це лямбда-функція, в ній один аргумент `x`, а результат — значення аргументу `x` у степені `n`.



Функція `my_pow()` під час виклику з одним аргументом (позначимо як `n`) повертає результатом функцію $f(x) = x^n$. Таким чином, інструкцію `my_pow(n)` можна розглядати як ім'я функції, якій передається один аргумент. Щоб обчислити значення x^n використовуємо інструкцію `my_pow(n)(x)`.

Для перевірки роботи створеної нами функції ми запускаємо вкладені оператори циклу, в яких перебираємо значення змінної `n` (від 1 до 3) і змінної `x` (від 1 до 10). Для кожної з пар значень команда `print(my_pow(n)(x),end=" ")` виводить (в один рядок — завдяки інструкції `end=" "`) значення виразу `my_pow(n)(x)`. Результат виконання програмного коду наведено нижче:

 **Результат виконання програми (з лістингу 3.10)**

```
1 2 3 4 5 6 7 8 9 10
1 4 9 16 25 36 49 64 81 100
1 8 27 64 125 216 343 512 729 1000
```



Щоб відображати значення в одному рядку, функції `print()` передається іменований аргумент `end=" "`. Для переходу до нового рядка використовуємо `print()` без аргументів.

Перший рядок значень — це натуральні числа від 1 до 10 у першому степені. Другий ряд — ті ж числа, але у другому степені. Третій ряд — третій степінь.



В описі лямбда-функцій для аргументів можна задавати значення за замовчуванням. Значення за замовчуванням указується після імені відповідного аргументу через знак рівності. Наприклад, інструкція `lambda x=1: 1/x` визначає лямбда-функцію для залежності $f(x) = \frac{1}{x}$ зі значенням аргументу за замовчуванням 1. Якщо цю лямбда-функцію присвоїти змінній (наприклад, виконати команду `f=lambda x=1: 1/x`), то потім під час виклику функції можемо аргумент вказувати (наприклад, `f(2)`) або не вказувати (наприклад, `f()`). В останньому прикладі використано значення аргументу за замовчуванням (у нашому випадку 1).

18.01.22

Локальні та глобальні змінні

Я теж побував за ґратами, але з поганого приводу.

Дж. Буш (молодший)

Ми вже описали чимало функцій і практично кожного разу використовували в тілі функцій ті або інші змінні. Особливих непорозумінь у нас не виникало. Однак тут є одна досить серйозна проблема, пов'язана з тим, де використовувані нами змінні доступні, а де — ні. Змінні, які доступні лише в тілі функції, називаються *локальними*. Змінні, доступні поза функцією, називаються *глобальними*.



У таких мовах програмування, як C++ і Java, де змінні оголошують із зазначенням типу, проблем із розподілом змінних на локальні та глобальні не виникає. Цю класифікацію легко провести на підставі того, де, в якому місці програмного коду змінну оголошено. У мові Python ситуація складніша. Річ у тім, що оскільки змінні в Python заздалегідь (до першого використання) не оголошують, то визначити, яка змінна локальна, а яка — глобальна, буває не так уже й просто.

Розглянемо деякі нескладні приклади, що ілюструють особливості використання локальних і глобальних змінних. Так, якщо в тілі функції змінній присвоєно значення, за замовчуванням ця змінна є локальною. Причому правило діє, навіть якщо раніше (до опису функції) було залучено глобальну змінну з таким же іменем. Саме такий випадок реалізовано в програмному коді в лістингу 3.11.

Лістинг 3.11. Локальна та глобальна змінні

```
# Глобальна змінна
x=100
# Опис функції
def test_vars():
    # Локальна змінна
    x="локальна змінна"
    # Перевіряємо значення змінної
    # у тілі функції
    print("У тілі функції x =", x)
# Викликаємо функцію
test_vars()
# Перевіряємо значення змінної
# поза тілом функції
print("Поза функцією x =", x)
```

На що тут варто звернути увагу: спочатку ми створюємо глобальну змінну `x` зі значенням 100. Потім оголошуємо функцію `test_vars()`, у тілі якої змінній `x` присвоюється текстове значення "локальна змінна". Команда `print("У тілі функції x =", x)` перевіряє значення змінної `x`. Це весь код функції.

Поза тілом функції виконується команда `test_vars()`, якою викликається описана вище функція, а потім командою `print("Поза функцією x =", x)` перевіряється значення змінної `x`. Результат виконання програмного коду наведено нижче:

Результат виконання програми (з лістингу 3.11)

У тілі функції x = локальна змінна
Поза функцією x = 100

Якщо за пунктами, то ситуація така: ми змінній присвоюємо значення, викликаємо функцію, в якій змінній із таким же іменем присвоюється значення (значення виводиться в консольне вікно), і після виклику функції знову перевіряємо значення змінної. Як видно з результату виконання програми, під час перевірки значення змінної `x` у тілі функції і значення змінної `x` поза тілом функції отримуємо різні результати. Що

Phyton

це означає? Означає те, що в тілі функції і поза тілом функції ми маємо справу з різними змінними, хоча в них і збігаються імена.

Якщо ми тепер трохи змінимо програмний код функції `test_vars()`, прибравши з нього команду присвоювання значення змінній `x` (але залишивши команду відображення значення цієї змінної), ситуація дещо зміниться. Новий програмний код наведено в лістингу 3.12.

Лістинг 3.12. Глобальні змінні

```
# Опис функції
def test_vars():
    # Перевіряємо значення змінної
    # у тілі функції. Значення змінній x
    # у тілі функції не присвоюється
    print("У тілі функції x =", x)
# Глобальна змінна
x="глобальна змінна"
# Викликаємо функцію
test_vars()
# Перевіряємо значення змінної
# поза тілом функції
print("Поза функцією x =", x)
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 3.12)

У тілі функції x = глобальна змінна

Поза функцією x = глобальна змінна

Тут ми в тілі функції виводимо на екран значення змінної `x`, але при цьому використовується те значення, що було присвоєно змінній `x` до виклику функції.



Зверніть увагу, що значення змінної `x` присвоюється після опису функції `test_vars()`, але до її виклику.

Таким чином, якщо в тілі функції є змінна, і цій змінній у тілі функції присвоюється значення, така змінна буде локальною. Якщо в тілі функції є змінна, але значення в тілі функції їй не присвоюється, то буде використано глобальну змінну (змінна, значення якої присвоєно поза тілом функції — але до її виклику).

У таку схему інтерпретації рівня доступу змінних іноді потрібно вносити зміни. Конкретніше, існує можливість у явному вигляді виділити (задекларувати) глобальні змінні в тілі функції. Для цього використовують ключове слово `global`. Після цього ключового слова перераховуються (через кому, якщо їх декілька) змінні, які слід інтерпретувати як глобальні. У листингу 3.13 наведено невеликий приклад.

 **Лістинг 3.13. Використання інструкції `global`**

```
# Глобальна змінна
x=100
# Опис функції
def test_vars():
    # Декларуємо глобальні змінні
    global x,y
    # Перевіряємо значення змінної x
    print("У тілі функції x =",x)
    # Значення глобальної змінної y
    y=200
    # Перевіряємо значення змінної y
    print("У тілі функції y =",y)
    # Значення глобальної змінної x
    x=300
    # Викликаємо функцію
test_vars()
# Перевіряємо значення змінної x
# поза тілом функції
print("Поза функцією x =",x)
# Перевіряємо значення змінної y
# поза тілом функції
print("Поза функцією y =",y)
```

Python

У цьому програмному коді використано інструкцію `global x, y`. Вона декларує змінні `x` і `y` як глобальні. Команда `print("У тілі функції x =", x)` у тілі функції перевіряє значення змінної `x`, а до цього, до оголошення функції, змінний `x` присвоєно значення 100. Саме це значення буде відображене у разі виконанні зазначененої вище команди.

Своє перше значення змінна `y` отримує в тілі функції (команда `y=200`). Потім ми перевіряємо значення цієї змінної (команда `print("У тілі функції y =", y)`). Нарешті, командою `x=300` присвоюється нове значення змінній `x` (яка, нагадаємо, є глобальною). На цьому опис функції закінчується.

Для перевірки роботи функції та ролі глобальних змінних викликаємо функцію `test_vars()`, а після цього перевіряємо значення змінних `x` і `y`. Результат виконання програми такий:



Результат виконання програми (з лістингу 3.13)

```
У тілі функції x = 100
У тілі функції y = 200
Поза функцією x = 300
Поза функцією y = 200
```

Загальний висновок може бути таким: якщо змінна глобальна, то зміна її значення також носить «глобальний» характер.

19.01.22

Вкладені функції

Я — компетентний орган. І мені вирішувати, що краще.

Дж. Буш (молодший)

У Python функції можна описувати всередині функцій. Такі функції будемо називати *вкладеними функціями* (або *внутрішніми функціями*). У цій ситуації було би мало цікавого, якби не одна особливість: вкладена функція має доступ до змінних зовнішньої функції (тієї функції, в якій описано вкладену функцію).

Причини, через які у програмному коді використовують вкладені функції, можуть бути різними. Наприклад, через вкладену функцію зручно реалізувати допоміжні обчислення. Інший варіант — вкладену функцію можна повернати як результат. Невеличкий приклад використання вкладених функцій наведено в лістингу 3.14. Там ми описуємо функцію для обчислення суми квадратів натуральних чисел. При цьому в тілі функції описано дві вкладені функції.

Лістинг 3.14. Використання вкладених функцій

```
# Зовнішня функція
def sq_sum():
    # Вкладена функція для зчитування
    # кількості доданків
    def get_n():
        # Зчитуємо числове значення
        n=int(input("Доданків у сумі: "))
        # Результат функції get_n() - ціле число
        return n
```

Phyton

```
# Вкладена функція для обчислення
# суми квадратів натуральних чисел.
# Результатом є функція
def find_sq_sum():
    # Початкове значення суми
    s=0
    # Оператор циклу для обчислення суми
    for i in range(1,n+1):
        s+=i**2 # Новий доданок у сумі
    # Результат функції find_sq_sum()
    return s
# Визначаємо кількість доданків у сумі
n=get_n()
# Результат функції sq_sum() - вкладена функція
return find_sq_sum
# Обчислюємо суму квадратів чисел
z=sq_sum()()
# Відображаємо результат
print("Сума квадратів дорівнює:", z)
```

У тілі зовнішньої функції, яка називається `sq_sum()` і призначена безпосередньо для обчислення суми квадратів натуральних чисел, описані вкладені функції `get_n()` і `find_sq_sum()`.

Під час виконання функції `get_n()` користувачеві пропонується ввести ціле число, яке визначає кількість доданків у сумі. Уведене користувачем значення повертається як результат функції `get_n()`. У тілі функції `get_n()` команда `n=int(input("Доданків у сумі: "))` відображає повідомлення з пропозицією ввести число і зчитує це число. Число записується в змінну `n`, яку команда `return n` повертає резултатом функції.



Для перетворення текстового значення, яке повертає функція `input()`, до ціличесового формату ми використовуємо функцію `int()`.

Вкладена функція для обчислення суми квадратів натуральних чисел `find_sq_sum()` містить такі команди. Команда `s=0` присвоює нульове

початкове значення змінній s , у яку записується сума квадратів чисел. Для обчислення суми запускається оператор циклу. В операторі циклу змінна i «пробігає» значення від 1 до n , і за кожен цикл виконується команда $s+=i**2$, яка до поточного значення суми s додає квадрат натурального числа $i**2$. Після закінчення оператора циклу значення змінної s повертається як результат функції.

Тут є один цікавий момент: в операторі циклу (в інструкції `range(1, n+1)`) використано змінну n , якій у тілі вкладеної функції `find_sq_sum()` значення не присвоєно. Значення змінної n визначається в тілі зовнішньої функції `sq_sum()`: там є команда `n=get_n()`, якою за допомогою виклику вкладеної функції `get_n()` визначається кількість доданків, і зчитане значення записується в змінну n . Після цього командою `return find_sq_sum` результатом функції `sq_sum()` повертається функція `find_sq_sum()`.

Отже, коли ми викликаємо функцію `sq_sum()`, у тілі цієї функції викликається функція `get_n()` і, як наслідок, з'являється повідомлення з пропозицією ввести числове значення. Це значення використовується у вкладеній функції `find_sq_sum()`, яка повертається як результат. Таким чином, вираз `sq_sum()` можемо інтерпретувати як змінну, яка посилається на функцію (тобто як ім'я функції). Якщо ми хочемо викликати цю функцію, необхідно додати ще одну пару дужок. Конкретніше, вираз `sq_sum()()` фактично є викликом функції `find_sq_sum()` (із попереднім викликом функції `get_n()`). Тому після виконання команди `z=sq_sum()()` у змінну z записується сума квадратів натуральних чисел. Отримане значення відображаємо за допомогою команди `print("Сума квадратів дорівнює:", z)`. Нижче показано результат виконання програми (жирним шрифтом виділено введене користувачем значення):



Результат виконання програми (з лістингу 3.14)

Доданків у сумі: **10**

Сума квадратів дорівнює: **385**

Як бачимо, результат вийшов коректний.

Функція як результат функції

Робота у мене така — думати далі
свого носа.

Дж. Буш (молодший)

Вище, розглядаючи лямбда-функції, ми вже познайомилися із ситуацією, коли функція як результат повертає функцію. На практиці такі задачі виникають досить часто. Відверто кажучи, можливі й інші, навіть більш екзотичні варіанти. Але екзотику ми залишимо «на потім», а тут ще раз повернемося до теми створення функцій, які повертають як свій результат функції. Цього разу розглянемо питання докладніше.

Важливий момент, який ми вже підкresлювали і який слід мати на увазі, — це те, що функція як така є деяким об'єктом. Тобто коли в програмному коді зустрічається опис функції, для цієї функції створюється об'єкт. Посилання на об'єкт записується в змінну — назву функції. У цьому відношенні важливо розуміти, що змінна, яка посилається на число, концептуально мало чим відрізняється від змінної, яка посилається на функцію (об'єкт функції). Якщо ми в будь-який момент можемо змінити значення «числової» змінної, то так само ми можемо змінити значення для змінної — імені функції. Достатньо цій змінній присвоїти нове значення — посилання на об'єкт іншої функції. Більше того, оскільки в Python тип для змінних не зазначається, і змінні можуть посылатися на будь-які значення, то теоретично ніхто не забороняє присвоїти змінній, яка раніше посылалася на функцію, посилання на дані іншого типу (наприклад, на числове значення або текст). Можлива і зворотна ситуація: змінна, яка раніше на функцію не посылалася, після присвоювання значення може «стати функцією». Це не питання можливості, а швидше питання доцільності.

Наприклад, якщо ми хочемо, щоб функція як результат повертала функцію, то теоретично можливі такі варіанти (найочевидніші):

- Використання лямбда-функції і повернення посилання на цю функцію. Такий підхід ми зустрічали раніше.
- Описання в тілі функції вкладеної функції і повернення посилання на цю функцію як результату зовнішньої функції.
- Як результат функції можливе повернення імені іншої функції (не вкладеної). Такий підхід теж можливий.

У лістингу 3.15 наведено приклад функції, яка як результат, залежно від переданого їй логічного аргументу, повертає посилання на функцію для обчислення факторіала або подвійного факторіала числа.



Нагадаємо, що факторіалом числа називається добуток усіх натуральних чисел, які не перевищують це число: $n! = n \cdot (n-1) \cdots \cdot 2 \cdot 1$. Подвійний факторіал числа — також добуток натуральних чисел, але тільки «через одне число»: $n!! = n \cdot (n-2) \cdot (n-4) \cdots$ (останній множник дорівнює 2 для парного числа n і дорівнює 1 — для непарного числа n).

Лістинг 3.15. Обчислення факторіала

```
# Функція для обчислення факторіала
# і подвійного факторіала
def factor(mode=True):
    # Вкладена функція для обчислення
    # факторіала числа
    def sf(n):
        s=1 # Початкове значення добутку
        i=n # Початкове значення індексу
        while i>1: # Умова
            s*=i # Множення на індекс
            i-=1 # Зменшення індексу на 1
        return s # Результат вкладеної функції
    # Вкладена функція для обчислення
    # подвійного факторіала числа
```

Phyton

```
def df(n):
    s=1 # Початкове значення добутку
    i=n # Початкове значення індексу
    while i>1: # Умова
        s*=i # Множення на індекс
        i-=2 # Зменшення індексу на 2
    return s # Результат вкладеної функції
# Якщо аргумент mode дорівнює True
if mode:
    return sf # Посилання на функцію для
                # обчислення факторіала
# Якщо аргумент mode дорівнює False
else:
    return df # Посилання на функцію для
                # обчислення подвійного факторіала
# Викликаємо функцію factor() для обчислення факторіала
print("5! =",factor()(5))
print("5! =",factor(True)(5))
# Викликаємо функцію factor() для обчислення
# подвійного факторіала
print("5!! =",factor(False)(5))
```

Результат виконання програмного коду такий:



Результат виконання програми (з листингу 3.15)

```
5! = 120
5! = 120
5!! = 15
```

Функцію `factor()` описано з одним аргументом `mode`, в якого є значення за замовчуванням `True`. Таким чином, функцію можна викликати без аргументів або з одним логічним аргументом. Якщо аргумент дорівнює `True`, функція як результат повертає функцію, призначенну для обчислення факторіала числа. Якщо функцію `factor()` викликано з логічним аргументом `False`, результатом буде функція, призначена для обчислення подвійного факторіала числа.

У тілі функції `factor()` описано дві вкладені функції: функція `sf()` обчислює факторіал, а функція `df()` обчислює подвійний факторіал. Залежно від значення аргументу `mode` в умовному операторі як результат функції `factor()` повертається посилання `sf` або `df`.



У Python існують більш прості способи обчислення добутку натуральних чисел, ніж створення для цієї мети оператора циклу. Однак ми до них принципово не вдаємося.

Під час перевірки функціональності створеної функції `factor()` нам на допомогу приходять команди `factor()(5)`, `factor(True)(5)` (в обох випадках ідеться про обчислення значення $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$) і `factor(False)(5)` (обчислення значення $5!! = 5 \cdot 3 \cdot 1 = 15$).



Зверніть увагу, що результатами виразів `factor()`, `factor(True)` і `factor(False)` є посилання на функції. Тому дані виразу слід розглядати як «назви» функцій. Щоб викликати ці функції, після «назв» у круглих дужках вказують аргумент функції (у цьому разі число 5). Звідси отримуємо відповідно `factor()(5)`, `factor(True)(5)` і `factor(False)(5)`.

Ще одне зауваження щодо розглянутої програми. Легко помітити, що програмні коди вкладених функцій `sf()` і `df()` практично ідентичні і відрізняються лише значенням декремента індексу в операторі циклу (величина, на яку зменшується змінна `i` — під час обчислення звичайного факторіала це 1, а під час обчислення подвійного факторіала індекс зменшується на 2). Ураховуючи цю обставину, розглянутий раніше програмний код можна було б організувати дещо інакше. Приклад наведено в лістингу 3.16.

Лістинг 3.16. Факторіал і подвійний факторіал

```
# Функція для обчислення факторіала
# і подвійного факторіала
def factor(mode=True):
    # Вкладена функція для обчислення
    # звичайного/подвійного факторіала числа
```

```
def f(n, d):  
    s=1 # Початкове значення добутку  
    i=n # Початкове значення індексу  
    while i>1: # Умова  
        s*=i # Множення на індекс  
        i-=d # Зменшення індексу  
    return s # Результат вкладеної функції  
  
# Значення декремента для індексу  
d=1 if mode else 2  
# Результат функції  
return lambda n: f(n, d) # Лямбда-функція  
  
# Викликаємо функцію factor() для обчислення факторіала  
print("5! =",factor()(5))  
print("5! =",factor(True)(5))  
# Викликаємо функцію factor() для обчислення  
# подвійного факторіала  
print("5!! =",factor(False)(5))
```

Результат виконання цього програмного коду такий самий, як і результат виконання програми з лістингу 3.15. Щодо самого програмного коду, то тепер у функції `factor()` описано лише одну вкладену функцію, яка називається `f()` і має два аргументи: число `n`, для якого обчислюється звичайний або подвійний факторіал, а також крок дискретності `d` для зменшення індексної змінної в операторі циклу. Фактично, значення аргументу `d` визначає, який факторіал (звичайний або подвійний) буде обчислено: для звичайного факторіала значення аргументу `d` дорівнює 1, а для обчислення подвійного факторіала значення цього аргументу повинно дорівнювати 2. У тілі функції `factor()` командою `d=1 if mode else 2` (використовується *тернарний оператор*) змінний `d` присвоюється значення 1 або 2 залежно від значення логічного аргументу `mode` функції `factor()`. Після цього як результат функції `factor()` повертає посилання на лямбда-функцію, задану виразом `lambda n: f(n, d)`. Головна ідея в тому, що вкладений функції під час виклику `f()` передаються два аргументи, в той час як функція `factor()` повинна повертати як результат функцію, в якої один аргумент. Тому повернати посилання на функцію `f()` як результат функції `factor()` — погане рішення. Ми зробимо інакше: створюємо анонімну функцію з одним аргументом, яка припускає виклик функції `f()` з цим же першим

аргументом, а другий аргумент функції `f()` — раніше визначене числове значення `d`.

Нарешті, ще один спосіб розв'язання тієї ж задачі проілюстровано в лістингу 3.17.

Лістинг 3.17. Ще один спосіб обчислити факторіал

```
# Функція для обчислення факторіала числа
def factorial(n):
    if n==1:
        return 1
    else: # Рекурсія
        return n*factorial(n-1)

# Функція для обчислення подвійного факторіала
def dfactorial(n):
    if n==1 or n==2:
        return n
    else: # Рекурсія
        return n*dfactorial(n-2)

# Функція для обчислення факторіала
# і подвійного факторіала
def factor(mode=True):
    # Результат - посилання на зовнішню функцію
    return factorial if mode else dfactorial

# Викликаємо функцію factor() для обчислення факторіала
print("5! =", factor()(5))
print("5! =", factor(True)(5))

# Викликаємо функцію factor() для обчислення
# подвійного факторіала
print("5!! =", factor(False)(5))
```

У програмному коді описано дві (зовнішні) функції: `factorial()` для обчислення факторіала числа й `dfactorial()` для обчислення подвійного факторіала числа. В описі цих функцій використовується рекурсія. При цьому програмний код функції `factor()` виключно простий і складається всього з однієї команди, яка повертає вираз `factorial if mode else dfactorial` як результат функції. Цей

вираз базується на *тернарному операторі*, і його значення дорівнює `factorial`, якщо аргумент `mode` набуває значення `True`. А якщо ні, результат виразу дорівнює `dfactorial`. В обох випадках ідеться про повернення посилання на зовнішню (відносно функції `factor()`) функцію. Результат виконання програмного коду такий самий, як і в попередніх ситуаціях.

Наступний приклад, який ми розглянемо, — це функція, якій аргументом передається функція, і яка результа том повертас функцію. Якщо конкретніше, то йтиметься про обчислення похідної.



Похідною від функції $f(x)$ називається деяка функція (позначають як $f'(x)$

або $\frac{df}{dx}$), яка дорівнює границі відношення приросту функції $f(x)$ до приросту аргументу Δx при прямуванні останнього до нуля:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

Для нас же важливо таке: існує правило (правило обчислення похідної), за яким одній функції можна у відповідність поставити іншу функцію (яка називається похідною). Наприклад, якщо $f(x) = x^2$,

$$\text{то } f'(x) = 2x, \text{ а якщо } f(x) = \frac{1}{1+x}, \text{ то } f'(x) = -\frac{1}{(1+x)^2}.$$

Узагалі процедура обчислення похідної — операція аналітична. Але на практиці часто вдаються до обчислення похідної в наближенному вигляді, для чого вибирається малий, але кінцевий, приріст аргументу Δx , і в точці x похідна

$$f'(x) \text{ обчислюється як відношення } f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

Саме таким підходом ми скористаємося для обчислення похідної в програмному коді.

У лістингу 3.18 наведено приклад програми, в якій описано функцію, що дозволяє обчислювати (у наближенному вигляді) похідну. Вихідна (диференційована) функція передається як аргумент, а похідна функція повертається як результат.



Лістинг 3.18. Обчислення похідної

```
# Функція для обчислення похідної
def D(f):
    # Вкладена функція. Обчислює
    # наближене значення похідної
```

```

def df(x, dx=0.001):
    # Результат вкладеної функції
    return (f(x+dx)-f(x))/dx
# Результат функції - похідна
return df

# Перша функція для диференціювання
def f1(x):
    return x**2

# Друга функція для диференціювання
def f2(x):
    return 1/(1+x)

# Функція для відображення похідної
# в декількох точках. Аргументи такі:
# F - похідна (наближена)
# Nmax - кількість точок (мінус один)
# Xmax - права границя по аргументу
# dx - приріст аргументу
# f - похідна (аналітична)
def show(F, Nmax, Xmax, dx, f):
    # Точки, в яких обчислюється похідна
    for i in range(Nmax+1):
        x=i*Xmax/Nmax # Значення аргументу
        # Наближене й точне значення похідної
        print(F(x), F(x,dx), f(x), sep=" -> ")

# Похідна для першої функції
F1=D(f1)
# Похідна для другої функції
F2=D(f2)
# Значення в різних точках
# похідної для першої функції
print("Похідна (x**2)'=2x:")
show(F1,5,1,0.01,lambda x: 2*x)
# Значення в різних точках
# похідної для другої функції
print("Похідна (1/(1+x))'=-1/(1+x)**2:")
show(F2,5,1,0.01,lambda x: -1/(1+x)**2)

```

Результат отримуємо такий:

■ Результат виконання програми (з лістингу 3.18)

Похідна $(x^{**2})' = 2x$:

```
0.001 -> 0.01 -> 0.0  
0.400999999999986 -> 0.4099999999999999 -> 0.4  
0.800999999999962 -> 0.8099999999999996 -> 0.8  
1.2010000000000076 -> 1.21 -> 1.2  
1.6009999999999636 -> 1.6100000000000003 -> 1.6  
2.0009999999996975 -> 2.0100000000000007 -> 2.0
```

Похідна $(1/(1+x))' = -1/(1+x)^{**2}$:

```
-0.9990009990008542 -> -0.990099009900991 -> -1.0  
-0.6938662225923764 -> -0.688705234159781 ->  
-0.694444444444444  
-0.5098399102682061 -> -0.5065856129686019 ->  
-0.5102040816326532  
-0.3903810118676132 -> -0.38819875776396895 ->  
-0.39062499999999994  
-0.3084706027516315 -> -0.30693677102516803 ->  
-0.30864197530864196  
-0.2498750624687629 -> -0.24875621890546595 -> -0.25
```

У функції `D()`, призначений для обчислення похідної, є один аргумент, який позначено як `f` і який ототожнюється з назвою диференційованої функції. У тілі функції `D()` описано вкладену функцію `df()` із двома аргументами. Перший аргумент `x` позначає точку, в якій обчислюється похідна. Другий аргумент `dx` позначає приріст по аргументу, на основі якого обчислюється вираз для похідної. Аргумент `dx` має значення за замовчуванням. У тілі вкладеної функції `df()` повертається (як результат функції) значення виразу $(f(x+dx) - f(x)) / dx$ (відношення приросту диференційованої функції до приросту аргументу). Функція `D()`, у свою чергу, повертає як результат посилання `df` на вкладену функцію `df()`.

Також у програмі описано дві функції для обчислення на їхній основі похідних. Ідеється про функцію $f1()$ (відповідає залежності $f(x) = x^2$) і функцію $f2()$ (відповідає залежності $f(x) = \frac{1}{1+x}$). Похідні для цих функцій розраховуємо командами $F1=D(f1)$ і $F2=D(f2)$ відповідно. Після виконання цих команд змінна $F1$ посилається на функцію, що відповідає похідній для функції $f1()$. Аналогічно, змінна $F2$ є посиланням на похідну для функції $f2()$. Причому функціям $F1()$ і $F2()$ під час виклику можна передавати один або два аргументи. Перший аргумент визначає точку, в якій обчислюється похідна, а другий аргумент, якщо його задано, визначає приріст по аргументу для обчислення похідної.

Для розрахунку й відображення значень похідних функцій у декількох точках використовуємо функцію $show()$. Функцію описано з такими аргументами:

- через F позначено функцію для обчислення похідної в числовому вигляді (на цій позиції під час виклику функції $show()$ вказуємо назви $F1$ і $F2$ функцій $f1()$ і $f2()$);
- змінна N_{\max} позначає кількість інтервалів, на які розбито діапазон зміни аргументу під час обчислення похідної (це число на одиницю менше від кількості точок, у яких обчислюється похідна);
- змінна x_{\max} позначає праву границю діапазону зміни аргументу (ліва границя дорівнює нулю);
- через dx позначено приріст аргументу;
- через f позначено ім'я функції, яка визначає аналітичне значення для похідної.

Під час виклику функції $show()$ останнім аргументом передаються лямбда-функції: $\lambda x: 2*x$ (відповідає залежності $f'(x) = 2x$) і $\lambda x: -1/(1+x)^2$ ($f'(x) = -\frac{1}{(1+x)^2}$). Під час виконання коду функції $show()$ відображаються три значення:

- значення похідної, обчислене на основі використованого за замовчуванням приросту аргументу (функції `F1()` і `F2()` викликають із одним аргументом);
- значення похідної, обчислене на основі явно вказаного значення для приросту аргументу (функції `F1()` і `F2()` викликають із двома аргументами);
- значення похідної, обчислене на основі аналітичного виразу.

Як бачимо, в основному, результати обчислення похідних числовими методами непогано корелюють із точними (обчисленими на основі аналітичних виразів) значеннями.

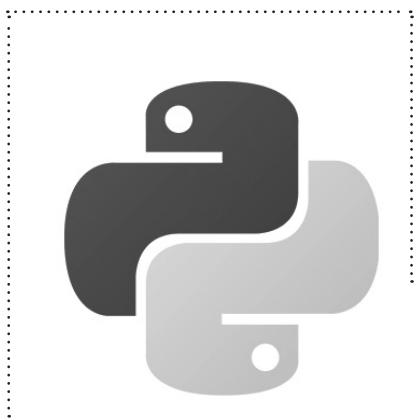
Резюме

Докази, які у мене були, — це були кращі з усіх можливих доказів.

Дж. Буш (молодший)

- В описі функції використовують ідентифікатор `def`, після якого вказують ім'я функції, список аргументів (у круглих дужках) і після двокрапки програмний код функції.
- Інструкція `return` у тілі функції закінчує виконання програмного коду функції, а значення, вказане після інструкції `return`, повертається як результат функції.
- Кожній функції відповідає об'єкт типу `function`. Ім'я функції є посиланням на об'єкт функції. Посилання на об'єкт функції може бути присвоєне змінній. У цьому разі змінна буде посиланням на функцію, і ця змінна може бути використана як ім'я функції.
- Ім'я функції може передаватися аргументом іншій функції.
- Функція може повертати як результат функцію. У цьому разі повертається посилання на функцію-результат.
- В аргументів можуть бути значення за замовчуванням. Значення аргументів за замовчуванням указують через знак рівності після імені аргументів. Аргументи зі значеннями за замовчуванням указують у списку аргументів функції останніми.
- В опису функції в тілі функції можна викликати описувану функцію (звичайно, з іншими аргументами). Така ситуація називається рекурсією.

- Лямбда-функція або анонімна функція — це функція без імені. Такі функції можна, наприклад, передавати аргументом в інші функції або повертати результатом функції. Описують лямбда-функцію за допомогою ключового слова `lambda`, після якого вказують аргументи і через двокрапку вираз, який є результатом лямбда-функції.
- Якщо змінній присвоєно значення в тілі функції, то така змінна є локальною. Вона доступна лише в тілі функції. Якщо змінна в тілі функції входить у вирази, але значення їй не присвоюється, то така змінна — глобальна. Щоб явно задекларувати змінну в тілі функції як глобальну, використовують ключове слово `global`.
- У тілі функції можуть бути описані й інші функції. Такі функції називають вкладеними. Вкладені функції мають доступ до змінних у тілі зовнішньої функції.



РОЗДІЛ 4

Робота зі списками і кортежами

Я намагаюся бути конкурентно-
здатною нацією.

Дж. Буш (молодший)

У цьому розділі ми ближче познайомимося зі *списками і кортежами*. Хоча, якщо чесно, то розділ в основному присвячено спискам. Із кортежами у нас буде досить поверхневе знайомство. Причина в тому, що кортежі, насправді, дуже «близькі» до списків, причому списки багато в чому є «більш гнучким» типом даних, порівняно з кортежами. Можна навіть сказати (через силу), що кортежі — це такі особливі списки. Тому ми спочатку познайомимося зі списками, визначимося з основними методами їхнього використання, а вже потім коротко зупинимося на кортежах.

Знайомство зі списками

Я — живий приклад того, хто бере проблему й отримує з неї вигоду.

Дж. Буш (молодший)

Хоча списки ми докладно не обговорювали, але нам уже доводилося з ними зустрічатися. Проте тоді, у попередніх розділах, ми в деталі особливо не вдавалися. Тепер настав час присвятити спискам більше уваги. І, треба сказати, вони на те заслуговують.

Список — це впорядкований набір елементів. У певному сенсі, списки в Python відіграють роль, аналогічну ролі масивів в інших мовах програмування. Але це тільки «у певному сенсі». Насправді, список у Python — це виключно гнучка, ефективна й десь навіть унікальна річ. Досить сказати, що елементами списку можуть бути об'єкти (дані) різного типу. Більше того, елементами списку, у свою чергу, можуть бути списки. Тобто за допомогою списків ми можемо створювати вкладену структуру практично будь-якої (в розумних межах, зрозуміло) складності.



Якщо ми говоримо про список як такий, то йдеться про тип `list`. Але у списку є елементи. І кожен із таких елементів може належати до якогось певного типу.

Як це було з числовими й текстовими значеннями, посилання на список записується в змінну. Таку змінну ми звичайно й будемо називати списком — якщо це не буде призводити до непорозумінь. Взагалі ж змінна *посилається* на список. Сам список створювати просто. Достатньо переврахувати елементи списку через кому, а всю цю послідовність елементів

узяти у квадратні дужки. Наприклад, команда `nums=[1, 2, 3]` створює список із трьох елементів, які є цілими числами (1, 2 і 3). Також можемо скористатися для створення списку функцією `list()`. У цьому разі як аргументи функції передаються елементи списку. Особливо зручно послуговуватися функцією `list()`, створюючи списки на основі текстових значень: якщо як аргумент функції `list()` передати текст, то в результаті отримаємо список, елементами якого будуть букви, що формують текст. Так, під час виконання команди `symbols=list("Python")` створюється список із букв P, y, t, h, o і n. Список може бути й більш «вищуканим»: скажімо, команда `data=["text", 100, [5,10]]` створює список `data` з трьох елементів, причому перший елемент списку — це текст "text", другий елемент списку - ціле число 100, а третій елемент — список із двох елементів [5, 10].



Створювати списки можна за допомогою спеціальних **генераторів списків**. У квадратних дужках указують вираз, що залежить від індексної змінної, яка «пробігає» певні значення. Діапазон зміни індексної змінної також задають у квадратних дужках. Наприклад, команда `pows_of_two=[2**i for i in range(11)]` створює список із десяти елементів, які є степенями двійки. Якщо після створення списку виконати команду `print(pows_of_two)`, отримаємо такий результат: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]. У такому разі під час формування списку елементи списку визначаються виразом 2^{**i} , і при цьому змінна i «пробігає» значення в діапазоні від 0 до 10 включно (інструкція `for i in range(11)`). Кожному індексу i відповідає елемент у списку, а значення елемента дорівнює 2^{**i} .

Крім діапазону зміни індексної змінної можемо зазначити умову, яка повинна виконуватися для внесення елемента до списку. Як ілюстрацію розглянемо команду `nums_list=[7*i+1 for i in range(20) if i%4==3]`, яка створює список [22, 50, 78, 106, 134] (аби побачити цей список, розумно скористатися командою `print(nums_list)`). Отриманий список складається з чисел, які при діленні на 7 дають остачу 1, а при діленні на 4 цілої частини від ділення на 7 дають остачу 3 (тобто якщо обчислити цілу частину від ділення числа на 7 і поділити її на 4, то в остачі буде 3). Як отримуємо список? Індекс i «пробігає» значення з діапазону від 0 до 19 включно, і якщо для поточного значення індексу i виконується умова $i\%4==3$ (остача від ділення індексу i на 4 дорівнює 3), то за формулою $7*i+1$ обчислюється черговий елемент списку.

Звертання до елемента списку відбувається за індексом елемента. Елементи індексуються, починаючи з нуля (тобто найперший елемент має нульовий індекс). Індекс указується у квадратних дужках після імені змінної, яка посилається на список. Якщо скористатися наведеними вище прикладами, то, скажімо, інструкція `nums[0]` є посиланням на перший елемент списку `nums` (тобто значення 1). Для звертання до третього (по порядку) елемента списку `symbols` (буква `t`) використовуємо інструкцію `symbols[2]`.

Звертаючись до елемента списку, можна зазначати від'ємний індекс — у цьому разі елементи «відраховуються», починаючи з кінця списку. Так, останній елемент буде мати індекс `-1`, передостанній елемент матиме індекс `-2` і так далі.



Визначити кількість елементів у списку дозволяє функція `len()`. Список передається як аргумент функції: наприклад, результатом виразу `len(symbols)` є значення 6 — кількість елементів у списку `symbols`. Аналогічно, результатом виразу `len(data)` буде значення 3, оскільки в списку `data` всього 3 елементи: (зверніть увагу: хоча останній елемент у списку `data` сам є списком, «рахується» він саме як один елемент).

Оскільки індексація елементів списку починається з нуля, то останній елемент списку має індекс, на одиницю менший, ніж довжина списку.

Список можна змінювати поелементно. Іншими словами, звертаючись до елемента списку, ми можемо не тільки «прочитати» значення елемента, але й змінити його. Наприклад, після виконання команди `nums[1]=-10` значення другого (по порядку) елемента списку `nums` дорівнюватиме `-10`, а сам список (який до цього був `[1, 2, 3]`) стане `[1, -10, 3]`.

Під час роботи зі списками ми можемо звертатися одразу до декількох елементів. У цьому випадку говорять про отримання *зрізу*. Фактично, йдеться про те, що ми беремо список і «видаємо» з нього деяку частину: усі елементи, індекси яких потрапляють у зазначеній діапазон. Для більшої конкретики уявімо, що в нас є деякий список. Якщо ми хочемо вилучити з цього списку групу елементів (підсписок) із індексами від `i`-го до `j`-го включно, то відповідна інструкція буде виглядати як `список[i:j+1]`. Тобто в тому місці, де ми раніше зазначали індекс,

тепер указано розділені двокрапкою два індекси, які й визначають елементи зрізу. Наприклад, значенням виразу `symbols[1:4]` є список із елементів `symbols[1]`, `symbols[2]` і `symbols[3]`, тобто список із букв `y`, `t` і `h`.



Зверніть увагу: якщо видаються елементи з індексами від i до j включно, то у квадратних дужках перед двокрапкою вказують індекс i першого елемента, який видають, а після двокрапки вказують індекс $j+1$ — тобто цей індекс на одиницю більший, ніж індекс останнього з елементів, які видають. Ще одне уточнення щодо змісту, яким ми наділяємо слово «видають»: важливо розуміти, що вихідний список не змінюється. Яким він був, таким і залишиться після процедури отримання зрізу.

Процедура отримання зрізу проста, зручна й часто дозволяє створювати ефективні програмні коди. Існує декілька правил, корисних для ефективного виконання процедури отримання зрізу. Для зручності сприйняття ми виділиммо основні формати, в яких може бути виконано зріз. Спочатку розглянемо команду отримання зрізу вигляду список [$i:j$], тобто коли після імені списку у квадратних дужках (через двокрапку) вказано два індекси, причому обидва ці індекси *невід'ємні*. У цьому разі є сенс пам'ятати, що:

- Результатом інструкції список [$i:j$] є список, що складається з елементів, індекси яких більші або дорівнюють i і менші від j . Наприклад, якщо `m=[1, 5, 10, 15, 20, 25]`, то результатом виразу `m[2:5]` є список `[10, 15, 20]` (елементи з 2-го по 4-й включно).
- Якщо індекси i або j перевищують довжину списку (кількість елементів у списку можна визначити за допомогою інструкції вигляду `len(список)`), то відповідний індекс (або індекси) неявно замінюється на значення `len(список)` (тобто дуже великі індекси автоматично «урізаються»). Так, для списку `m=[1, 5, 10, 15, 20, 25]` результатом виразу `m[2:100]` буде значення `[10, 15, 20, 25]`, оскільки під час обчислень другий індекс 100 автоматично замінюється на значення 6 (значення виразу `len(m)` дорівнює 6).
- Якщо не вказати перший індекс i , він, за замовчуванням, уважається нульовим: наприклад, команда список [$:j$] еквівалентна команді список [$0:j$]. Для списку `m` команда `m[:4]`

Розділ 4. Робота зі списками і кортежами

- повертає список елементів із індексами від 0 до 3 включно (отримуємо список `[1, 5, 10, 15]`, як для команди `m[0:4]`).
- Якщо не вказати другий індекс `j`, то він, за замовчуванням, дорівнює довжині списку: наприклад, команда `список[i:]` еквівалентна команді `список[i:len(список)]`. Для списку `m` результатом виразу `m[3:]` буде список `[15, 20, 25]` (такий же результат отримуємо як значення інструкції `m[3:6]`).
 - Якщо перший індекс більший, ніж другий індекс, або дорівнює йому, зріз буде порожнім (отримуємо список без елементів). Наприклад, як результат вираз `m[4:3]` повертає порожній список (тобто `[]`).



Таким чином, копію списку можна отримати за допомогою інструкції `список[:]`.

У команді формату `список[i:j]` один із індексів або навіть обидва можуть бути від'ємними. Якщо перший `i`/або другий індекс від'ємні, то щоб зрозуміти, як оброблюється така інструкція, слід замінити від'ємний індекс на `len(список)+індекс` (але при цьому індекс -0 означає індекс 0). Скажімо, якщо ми маємо справу зі списком `m=[1, 5, 10, 15, 20, 25]`, то команда `m[1:-2]` еквівалентна команді `m[1:4]`, оскільки значення -2 для індексу `j` еквівалентне значенню 4 (довжина списку 6 плюс формальне значення індексу -2, що в результаті дає 4). Далі, за тією ж схемою легко можемо показати, що команда `m[-5:-2]` еквівалентна команді `m[1:4]`.



Від'ємний індекс, таким чином, можемо інтерпретувати як порядковий номер елемента у списку, якщо рахувати елементи з кінця. Правда, така інтерпретація від'ємних індексів не завжди вдала для розуміння техніки обчислень.

Крім того, якщо вказано «надто від'ємний» індекс (тобто від'ємний індекс, який за модулем перевищує довжину списку), для такого індексу використовується нульове значення.

Під час отримання зрізу можна вказати не два, а три індекси. У цьому разі третій індекс (який, до речі, може бути від'ємним) визначає пріоріт

для індексу елемента, що включається у зріз. Ідеється про інструкції вигляду `список[i:j:k]`. У результаті отримуємо зріз — список елементів із індексами i , $i+k$, $i+2*k$, $i+3*k$ і так далі (але останній індекс не більший, ніж j). Наприклад, для списку $m=[1, 5, 10, 15, 20, 25]$ результатом виразу $m[1:5:2]$ є список-зріз елементів з індексами 1 і 3 (тобто список $[5, 15]$).

Якщо команду отримання зрізу використано у форматі з трьома індексами (тобто якщо ми намагаємося отримати зріз командою вигляду `список[i:j:k]`) і третій індекс (k) додатний, то справедливими залишаються наведені вище правила інтерпретації відсутніх індексів, від'ємних індексів й індексів, які перевищують значення індексу останнього елемента. Але якщо третій індекс k від'ємний, то зрозуміти, за якими правилами отримано зріз, може бути не так вже й просто. Головне правило, яке слід запам'ятати: якщо в інструкції отримання зрізу третій індекс від'ємний, то елементи зі списку вибираються до зрізу у зворотному порядку, тобто справа наліво. При цьому перший індекс i повинен бути таким, щоб елемент, який відповідає цьому індексу, знаходився праворуч від елемента, який відповідає індексу j . Якщо цю умову не виконано, то в результаті отримуємо порожній список.

У разі якщо за від'ємного третього індексу (мається на увазі індекс k) не вказано перший індекс i , то в цій «позиції» використовується індекс останнього елемента в списку. Якщо ж не вказано другий індекс j , то перебір елементів виконується до початкового елемента включно.



Щодо третього індексу k (у команді вигляду `список[i:j:k]`), то його значення не може бути нульовим, а якщо цей індекс не вказаний (команда вигляду `список[i:j:]` або `список[:j]`), то, за замовчуванням, використовується одиничне значення.

Ситуація з виконанням зрізу командою вигляду `список[i:j:k]` із від'ємним третім індексом k може здатися трохи заплутаною. Тому для більшої наочності наведемо кілька простих прикладів виконання зрізу у форматі команди з трьома індексами. Зріз будемо отримувати на основі списку $m=[1, 5, 10, 15, 20, 25]$. Результати низки команд із поясненнями наведено в таблиці 4.1.

Розділ 4. Робота зі списками і кортежами

Таблиця 4.1. Виконання зрізу

Команда	Результат	Пояснення
<code>m[5:1:-2]</code>	<code>[25, 15]</code>	Вибираються елементи з 5-го по 1-й індекс із кроком два, причому елемент із індексом 1 до списку-результату не включається. У підсумку отримуємо список елементів із індексами 5 і 3
<code>m[5:-1:-1]</code>	<code>[]</code>	Хоча перший індекс формально більший, ніж другий, другий індекс від'ємний (значення <code>-1</code>) і відповідає останньому елементу в списку <code>m</code> . Перший індекс (значення <code>5</code>) також відповідає останньому елементу в списку <code>m</code> . Тому результат зрізу — порожній список
<code>m[-5:-1:-1]</code>	<code>[]</code>	Індекс <code>-5</code> відповідає другому зліва елементу в списку <code>m</code> . Індекс <code>-1</code> відповідає останньому (шостому зліва) елементу в списку <code>m</code> . Оскільки третій індекс (крок приросту) від'ємний, у результаті виконання зрізу отримуємо порожній список
<code>m[-1:-5:-1]</code>	<code>[25, 20, 15, 10]</code>	Зріз утворюється шляхом вибору зі списку <code>m</code> елементів від останнього (індекс <code>-1</code>) до третього зліва (індекс <code>-4</code>) з одиничним кроком зменшення індексу. Зверніть увагу, що елемент із індексом <code>-5</code> до списку-результату не включається

Команда	Результат	Пояснення
<code>m[::-1]</code>	[25, 20, 15, 10, 5, 1]	Не вказано перший і другий індекси, а третій індекс дорівнює -1. У цьому разі перебір елементів виконується від останнього елемента до першого (включно!) з одиничним кроком (тобто елементи списку включаються до зрізу у зворотному порядку)
<code>m[:-10:-1]</code>	[25, 20, 15, 10, 5, 1]	Перший індекс не вказано, а другий індекс -10 виходить за діапазон значень індексів елементів списку m. У результаті вибірка елементів для зрізу виконується (з одиничним кроком зменшення індексу), починаючи з останнього елемента й закінчуючи першим елементом списку m
<code>m[10:-10:-1]</code>	[25, 20, 15, 10, 5, 1]	Перший і другий індекси виходять за діапазон реальних значень індексів елементів списку m. У цьому разі під час обчислення зрізу вибираються всі елементи списку m (у зворотному порядку, від останнього до першого включно)

Розділ 4. Робота зі списками і кортежами

Команда	Результат	Пояснення
<code>m[10:0:-1]</code>	[25, 20, 15, 10, 5]	Значення першого індексу надто велике (у списку <code>m</code> усього 6 елементів). У цьому разі ефект такий самий, якби було б указано індекс останнього елемента списку <code>m</code> . Другий індекс нульовий. Оскільки третій індекс від'ємний (значення <code>-1</code>), то при отриманні зразу вибираються елементи, починаючи з останнього, з одиничним кроком зменшення індексу. Індекси елементів, які вибираються, повинні бути більшими за нуль. Як наслідок, перший елемент списку <code>m</code> (елемент із нульовим індексом) до зразу не потрапляє



Вище ми бачили, наскільки різноманітними можуть бути індекси під час отримання зразу. Але важливо розуміти, що в усіх цих випадках ішлося про **інтерпретацію** різних числових значень за використання їх як індексів. Щодо списків, то нагадаємо: індексація елементів списку починається з нуля, а індекс останнього елемента в списку на одиницю менший від довжини списку (кількість елементів у списку).

Далі ми розглянемо деякі прості операції, які виконуються зі списками.

Основні операції зі списками

Це важливо для людей — знати, що я президент кожного.

Дж. Буш (молодший)

Зрозуміло, що списки в програмі створюються не просто так, а для чогось. Так для чого ж створюються списки, і як вони використовуються? Щоб відповісти на ці запитання, є сенс розглянути ті операції, які можуть виконуватися зі списками. Поки що ми знаємо не дуже багато: як створити список, як звернутися до елемента списку і як отримати зріз. Останню операцію, вочевидь, можна розглядати як розширеній варіант процедури звернення до елемента списку. Але разом із тим, природно виникає ряд запитань. Наприклад, як додати або видалити елемент зі списку? Як копіювати списки? Звичайно, запитань може бути набагато більше, але спочатку розберімося хоча б із цими.

Для додавання елемента в кінець списку використовують метод `append()`. Метод викликають зі списку (формат команди `список.append(аргумент)`). Як аргумент методу передається значення (змінна), яку додають до списку. Припустімо, що в нас є список `s=[10, 20, 30]`. Щоб додати в кінець цього списку новий елемент (nehай для зручності це буде число 100), використовуємо команду `s.append(100)`. Якщо після цього перевірити значення списку `s` (скриставшись командою `print(s)`), отримаємо список `[10, 20, 30, 100]`.



Метод багато в чому схожий на функцію, тільки функція існує сама по собі, а метод «прив'язаний» до деякого об'єкта й викликається з цього об'єкта. У даному разі в ролі об'єкта виступає список. Якщо ми говоримо про

Розділ 4. Робота зі списками і кортежами

метод `append()`, то в кожного списку є свій метод із такою назвою. Під час виклику методу потрібно вказати не тільки ім'я методу, який викликають, але й список, для якого викликають метод. При цьому використовують «крапковий» синтаксис: після імені списку через крапку зазначають ім'я методу, який викликають. Метод, який викликають, має доступ до списку, з якого його викликають. Тому, наприклад, метод `append()` може змінити той список, із якого був викликаний (додати до списку елемент).

Схожим чином діє й метод `extend()`, але цей метод призначено для додавання в кінець списку групи елементів. Групу елементів, що додають, у свою чергу, зазвичай також реалізують у вигляді списку. Іншими словами, за допомогою методу `extend()` у кінець списку може бути додано інший список, причому додають його не як окремий елемент, а поелементно: кожен елемент списку, що додається, стає елементом вихідного списку (того, з якого викликають метод `extend()`). У цьому разі будемо говорити про **розширення** списку.



Якщо ми до списку A додаємо список B за допомогою методу `append()` (команда `A.append(B)`), то до списку A буде додано новий елемент – список B (увесь список B входить до списку A як один елемент). Якщо ми в список A додаємо список B за допомогою методу `extend()` (команда `A.extend(B)`), то до списку A буде додано елементи списку B.

Як ілюстрацію розглянемо невеликий програмний код, наведений у лістингу 4.1.

Лістинг 4.1. Додавання елементів до списку

```
# Базовий список
s=[10,20,30]
# Додаємо до списку новий
# елемент – список
s.append([1,2])
# Перевіряємо результат
print(s)
# Розширюємо список за рахунок
# іншого списку
s.extend([3,4])
```

Phyton

```
# Перевіряємо результат  
print(s)
```

Результат виконання цього програмного коду наведено нижче:

■ Результат виконання програми (з лістингу 4.1)

```
[10, 20, 30, [1, 2]]  
[10, 20, 30, [1, 2], 3, 4]
```

У цьому разі створюємо список `s=[10,20,30]` із трьох елементів. Коли виконується команда `s.append([1,2])`, до списку `s` додається новий елемент, і цей елемент — список `[1,2]`. Результатом є список `[10,20,30,[1,2]]`. Тобто якщо ми додаємо список до іншого списку, то список, який додають, стає новим елементом у тому списку, до якого його додають. Коли ж ми розширюємо список `s` командою `s.extend([3,4])`, то список `[3,4]` додається до списку `s` не як окремий елемент, а у вигляді окремих елементів. У результаті (з урахуванням унесених раніше до списку `s` змін) отримуємо список `[10,20,30,[1,2],3,4]`.

Використовуючи функцію `append()` або функцію `extend()`, елемент або елементи додають у кінець списку. Метод `insert()` дозволяє вставити елемент у список у тій позиції, яку ми визначаємо самі. Як перший аргумент методу передається індекс елемента, що додається до списку. Фактично, це той індекс, який матиме елемент, доданий до списку. Елементи, які вже були у списку, зсуванняться на одну позицію праворуч. Тобто той елемент, який у списку перебував на позиції, на яку додається новий елемент, не губиться, а зсуванняться праворуч. Як другий аргумент методу `insert()` передається власне елемент, який додається до списку. Ще раз зауважимо, що за допомогою методу `insert()` до списку можна додати тільки один елемент. Якщо нам потрібно додати декілька елементів до списку (за принципом, як це робиться методом `extend()`, коли елементи, які додаються, самі організовані у вигляді списку), можемо скористатися інструкцією присвоювання значення зрізу. Зокрема, якщо ми маємо справу зі списком, і нам потрібно на позицію, яка відповідає `i`-му індексу, вставити деяке значення (мається на увазі вставка елемента, а не присвоювання елементу списку нового значення), то відповідна команда буде виглядати як список `[i:i]=значення`. Іншими словами, зліва від оператора присвоювання розташовується інструкція зрізу, в якій

через двокрапку вказують один і той же індекс — цей індекс і визначає позицію в списку, на яку виконується вставка значення. Якщо ж ми скористаємося командою вигляду список`[i:i+1]=значення`, то буде виконано не вставку нового елемента в список, а зміна значення елемента списку з індексом `i`.



Якби ми захотіли отримати зріз деякого списку за допомогою команди вигляду список`[i:i]`, то отримали би порожній список, оскільки в цьому разі перший і другий індекси збігаються. Команда отримання зрізу виду список`[i:i+1]` — це насправді список із одного елемента з індексом `i`. Таким чином, якщо ми присвоюємо значення виразу список`[i:i]`, то начебто виходить, що присвоюємо значення елементу, якого немає. У результаті такий елемент у списку з'являється. Якщо ми присвоюємо значення виразу список`[i:i+1]`, то значення присвоюється елементу списку з індексом `i`. Більше того, як ми побачимо далі, якщо присвоїти деяке значення зрізу список`[i:i+k]`, то це приведе до того, що в списку буде видалено елементи з індексами від `i` до `i+k-1` включно і замість них додано елементи зі списку, який присвоюється як значення.

Деякі приклади вставки нових елементів у список наведено в лістингу 4.2.

Лістинг 4.2. Вставка елементів у список

```
# Вихідний список
s=[10,20,30]
# До списку додаємо елемент
# (число -5 другим елементом зліва в списку)
s.insert(1,-5)
# Перевіряємо результат
print(s)
# До списку додаємо елемент - список
# (другий елемент зліва - список [1,2])
s.insert(1,[1,2])
# Перевіряємо результат
print(s)
# До списку додаємо два елементи
# (реалізовані у вигляді списку з двох елементів)
s[2:2]=[3,4]
```

```
Phyton .....  
# Перевіряємо результат  
print(s)  
# Елементу списку присвоюється значення  
# (список [100,200] - значення третього елемента)  
s[2:3]=[100,200]  
# Перевіряємо результат  
print(s)  
.....
```

Результат виконання цієї програми такий:

Результат виконання програми (з лістингу 4.2)

```
[10, -5, 20, 30]  
[10, [1, 2], -5, 20, 30]  
[10, [1, 2], 3, 4, -5, 20, 30]  
[10, [1, 2], 100, 200, 4, -5, 20, 30]
```

Як вихідний ми використовуємо список `s=[10,20,30]`. Щоб до цього списку на позицію з індексом 1 (друга позиція зліва) додати новий елемент (число `-5`) використовуємо команду `s.insert(1,-5)`, у якій метод `insert()` викликається зі списку `s`, а аргументами методу передаються індекс 1 (місце в списку `s` для вставки елемента) і значення `-5` (елемент, який додається до списку `s`). Після цього командою `print(s)` відображається вміст списку `s`. У цьому разі отримуємо результат `[10,-5,20,30]`. Що ми бачимо? До списку `s`, який спочатку складався з трьох чисел `[10,20,30]`, на позицію з індексом 1 (другий елемент зліва) додано числовой елемент `-5`, а елементи списку `20` і `30` зсунулися на одну позицію вправо. Далі виконується команда `s.insert(1,[1,2])`. У результаті на позицію з індексом 1 поміщається новий елемент — список `[1,2]`. Цей список, як елемент списку `s`, розташовується на другій позиції зліва (індекс 1), а інші три елементи списку `s` зміщуються вправо. Внаслідок цієї операції список `s` стає таким: `[10,[1,2],-5,20,30]`.

Наступним кроком ми додаємо до списку `s` два елементи. Для цього використовуємо команду `s[2:2]=[3,4]`. Вставка відбувається на позицію з індексом 2 (третій елемент зліва в списку `s`). Але тепер до списку `s` додається не список-елемент, а два елементи — ті елементи, які формують список `[3,4]`. Тому отримуємо список `[10,[1,2],3,4,-5,20,30]`. Нарешті, після виконання команди `s[2:3]=[100,200]` третій елемент

Розділ 4. Робота зі списками і кортежами

списку `s` (елемент із індексом 2) змінює своє значення з 3 на `[100, 200]`. У підсумку отримуємо список `[10, [1, 2], 100, 200, 4, -5, 20, 30]`.



Методи `append()`, `extend()` і `insert()` змінюють список, із якого викликаються, і результат не повертають.

Для об'єднання списків можна використовувати й оператор додавання. Наприклад, результатом виразу `[1, 2, 3] + [4, 5]` є список `[1, 2, 3, 4, 5]`.

Для видалення елемента зі списку використовують метод `pop()`. Аргументом методу передається індекс елемента, який потрібно видалити. Іншими словами, метод `pop()` дозволяє видалити елемент списку з певним індексом. Метод `pop()` не тільки змінює список, із якого викликається (у списку видаляється елемент), а ще й повертає результат — значення елемента, що видаляється. Якщо потрібно видалити елемент списку з певним значенням, то використовують метод `remove()`. Як аргумент методу передається значення елемента, який потрібно видалити зі списку. Якщо в списку декілька елементів мають таке значення, видаляється перший елемент зліва. Метод `remove()` змінює список, із якого викликається, і не повертає результат.

Ще один прийом, який використовують для видалення елемента або елементів зі списку, полягає в тому, що зrzізу списку значенням присвоюється порожній список. Наприклад, якщо ми хочемо в деякому списку видалити всі елементи з індексами від `i` до `j` включно, то відповідна команда могла б виглядати як список `[i:j+1] = []`.



Нагадаємо, що якщо присвоїти зrzізу непорожній список, елементи зі зrzізу у вихідному списку будуть видалені. Замість них буде вставлено елементи з того списку, який присвоюється як значення зrzізу.

Видалити елемент зі списку можна за допомогою інструкції `del` — тієї ж інструкції, якою видаляють із пам'яті змінні. Тільки якщо для видалення змінної її ім'я потрібно вказати після інструкції `del`, то для видалення елемента зі списку після інструкції `del` вказують посилання на цей елемент (у форматі список[індекс]). Наприклад, якщо ми

хочемо видалити в списку *s* елемент із індексом *i*, команда для видалення цього елемента виглядатиме як `del s[i]`. Невеличкий приклад із видаленням і заміною елементів у списку наведено в лістингу 4.3.

Лістинг 4.3. Видалення елементів зі списку

```
# Створюємо вихідний список
s=[i*(10-i) for i in range(11)]
# Перевіряємо результат
print(s)
# Видаляємо елемент із індексом 5
# і відображаємо значення елемента
print("Видаляємо елемент",s.pop(5))
# Перевіряємо вміст списку
print(s)
# Видаляємо елемент зі значенням 21
s.remove(21)
# Перевіряємо вміст списку
print(s)
# Видаляємо зі списку елемент
# із індексом 3
del s[3]
# Перевіряємо вміст списку
print(s)
# Видаляємо декілька елементів у списку
s[2:5]=[]
# Перевіряємо вміст списку
print(s)
# У списку видаляється два елементи
# і додається п'ять елементів
s[1:3]=[-1,-2,-3,-4,-5]
# Перевіряємо вміст списку
print(s)
```

Результат виконання програмного коду такий:

■ Результат виконання програми (з листингу 4.3)

```
[0, 9, 16, 21, 24, 25, 24, 21, 16, 9, 0]
```

Видаляємо елемент 25

```
[0, 9, 16, 21, 24, 24, 21, 16, 9, 0]
```

```
[0, 9, 16, 24, 24, 21, 16, 9, 0]
```

```
[0, 9, 16, 24, 21, 16, 9, 0]
```

```
[0, 9, 16, 9, 0]
```

```
[0, -1, -2, -3, -4, -5, 9, 0]
```

Спочатку за допомогою команди `s=[i*(10-i) for i in range(11)]` ми створюємо список із числових значень (індексна змінна `i` «пробігає» значення від 0 до 10, а відповідний елемент списку визначається виразом `i*(10-i)`). У результаті отримуємо список `[0, 9, 16, 21, 24, 25, 24, 21, 16, 9, 0]`.

Командою `print("Видаляємо елемент", s.pop(5))`, по-перше, видаляється елемент із індексом 5 і, по-друге, значення цього елемента відображається в консольному вікні. Тут мискористалися тим, що під час виконання інструкції `s.pop(5)` зі списку `s` видаляється елемент із індексом 5, а значення цієї інструкції — це значення елемента, який видаляється (у цьому разі йдеться про елемент зі значенням 25). Результатом виконання наведеної вище команди є повідомлення Видаляємо елемент 25, а сам список `s` при цьому буде таким: `[0, 9, 16, 21, 24, 24, 21, 16, 9, 0]` (тобто в ньому дійсно видалено елемент зі значенням 25).

За допомогою команди `s.remove(21)` видаляємо елемент зі значенням 21. Після цього список `s` стане таким: `[0, 9, 16, 24, 24, 21, 16, 9, 0]`. Важливим є те, що до видалення у списку `s` було два елементи зі значенням 21. Під час видалення елемента видаляється той, який у списку перший. А другий елемент зі значенням 21 у списку `s` залишається.

Командою `del s[3]` зі списку `s` видаляємо елемент із індексом 3. Після цього список `s` такий: `[0, 9, 16, 24, 21, 16, 9, 0]`. Бачимо, що список утратив один елемент зі значенням 24 (той, який був на позиції з індексом 3).

Щоб видалити елементи з індексами від 2-го до 4-го включно, використовуємо команду `s[2:5]=[]`. Тут зрізу присвоєно як значення порожній список. Список `s` перетворюється на `[0, 9, 16, 9, 0]`.

Команда `s[1:3]=[-1, -2, -3, -4, -5]` ілюструє операцію вставки (із заміною) до списку декількох елементів. Видаляються елементи списку `s` із індексами 1 і 2, а замість них уstawляється група елементів зі списку `[0, -1, -2, -3, -4, -5, 9, 0]`.

Щоб перевірити, чи входить елемент до списку, використовують оператор `in`. Результатом виразу формату значення `in` список є логічне значення `True` або `False` залежно від того, входить значення в список чи ні. Наприклад, результатом виразу `2 in [1, 2, 3]` є значення `True`, оскільки в списку `[1, 2, 3]` представлено значення 2, а результатом виразу `0 in [1, 2, 3]` є значення `False`, оскільки в списку `[1, 2, 3]` значення 0 немає.

За допомогою оператора `in` можна отримати відповідь на запитання, чи входить значення до списку. Якщо потрібно дізнатися, де саме у списку (на якій позиції) перебуває значення в списку, використовують метод `index()`. Метод викликається зі списку. Як аргумент методу передається значення, яке перевіряють на предмет входження до списку. Результатом є індекс елемента з відповідним значенням. Якщо значення, яке перевіряють, мають одразу декілька елементів, повертається індекс першого елемента з початку списку. Наприклад, значенням виразу `[1, 2, 3, 2, 1].index(2)` є 1, оскільки в списку `[1, 2, 3, 2, 1]` перший елемент зі значенням 2 розташований на позиції з індексом 1.



Звертаємо увагу читача й на те, як вище викликано метод `index()`. А саме, ми викликали цей метод безпосередньо зі списку, без попереднього приєднання списку значенням змінній. Такий підхід припустимий, однак не завжди віправданий. Зокрема, метод `index()` не змінює список, із якого викликається. Тому в конкретній ситуації наш підхід прийнятний.

Методу `index()` як другий аргумент можна передати значення індексу, з якого в списку починається пошук. Наприклад, значенням виразу `[1, 2, 3, 4, 3, 2, 1].index(2, 3)` є 5. Пошук елемента зі значенням 2 починається в списку `[1, 2, 3, 4, 3, 2, 1]` із елемента з індексом 3. Тому

Розділ 4. Робота зі списками і кортежами

перша двійка в списку в область пошуку не потрапляє, і в результаті повертається індекс 5 того елемента, значення якого дорівнює 2, але який переворує праворуч від елемента з індексом 3 (сам елемент із індексом 3 також потрапляє в діапазон пошуку). Методу `index()` також припустимо передати третій аргумент, який визначає кінцеву границю діапазону пошуку елемента. Слід урахувати, що якщо елемент із вказаним значенням не знайдено, виникає помилка.

Метод `count()` дозволяє порахувати кількість елементів у списку з певним значенням. Він викликається зі списку, а як аргумент методу передається значення елементів для пошуку. Наприклад, значення виразу `[1,2,3,2,1,3,2].count(2)` дорівнює 3, оскільки в списку `[1,2,3,2,1,3,2]` є три елементи зі значенням 2.

Метод `reverse()` і функція `reversed()` дозволяють змінити порядок елементів у списку на протилежний. Відмінність між методом і функцією в тому, що:

- метод `reverse()` (без аргументів) викликають зі списку, він не повертає результат і змінює список (із якого його викликають);
- під час виклику функції `reversed()` список передається їй як аргумент, причому сам список не змінюється, а як результат функції повертається новий список зі зворотним порядком елементів (порівняно зі списком, переданим аргументом функції).

Аналогічна ситуація з методом `sort()` і функцією `sorted()`. Метод і функція призначенні для сортування елементів списку:

- метод `sort()` викликають зі списку, який власне й сортують — у порядку збільшення, якщо метод викликано без аргументів, і в порядку зменшення, якщо метод викликано зі значенням аргументу `reverse=True` (аргумент звичайно передається за ключем);
- функції `sorted()` передається вихідний список, а результатом є відсортований список (сортування в порядку збільшення, а для сортування в порядку зменшення функції передають ще один аргумент `reverse=True`).

Існують й інші дуже корисні функції й методи, які широко застосовують під час роботи зі списками (і не тільки). Знайомитися з ними будемо за необхідності.



Є ряд математичних функцій, призначених для роботи зі списками. Наприклад, функції `min()` і `max()` призначені відповідно для визначення мінімального й максимального значень зі списку значень (спісок передається як аргумент функції). Наприклад, значення виразу `min([3, 4, -1, 7, 0])` дорівнює `-1` (найменше зі значень, наведених у списку `[3, 4, -1, 7, 0]`), а значення виразу `max([3, 4, -1, 7, 0])` дорівнює `7` (найбільше зі значень, наведених у списку `[3, 4, -1, 7, 0]`). Функцію `sum()` застосовують для обчислення суми значень списку, переданого аргументом функції: значення виразу `sum([3, 4, -1, 7, 0])` дорівнює `13` (сума чисел у списку `[3, 4, -1, 7, 0]`).

Копіювання і присвоювання списків

Я походжу від іншого покоління моого батька.

Дж. Буш (молодший)

Є одна, на перший погляд, проста задача, яка має серйозні «наслідки». Ця задача — створення копії списку. Формально тут все просто: на основі вже існуючого списку створюють достату такий же. Чому ж така задача має наслідки? Тому що «механізми», які задіяні або проявляються під час розв'язання цієї задачі, дають ключ до розуміння багатьох важливих моментів, пов'язаних із програмуванням в Python. Щоб не бути голослівними, одразу звернемося до прикладу.

Припустімо, що ми маємо список, наприклад, такий: `a=[10, 20, 30]`. Після цього виконуємо команду `b=a`. Відповідно змінній `b` як значення присвоюється значення змінної `a`, яка, у свою чергу, посилається на список. Логічно думати, що змінна `b` теж буде посилатися на список — та-кий самий, як і список, на який посилається змінна `a`. У тому, що списки однакові, переконатися легко. Для цього можемо скористатися, наприклад, командами `print(a)` і `print(b)`. Результати будуть однаковими (відобразиться один і той самий список `[10, 20, 30]`). Але ситуація не така проста, як видається на перший погляд. Припустімо, командою `b[1]=0` ми змінили значення другого (з індексом 1) елемента списку `b`. Якщо після цього командою `print(a)` ми перевіримо значення списку `a`, то отримаємо результат `[10, 0, 30]`. Іншими словами, зміни ми вносили у список `b`, а змінили при цьому список `a` (список `b` теж змінився — хто бажає, може в цьому легко переконатися). Пояснення таке: під час виконання команди `b=a` змінна `b` отримує посилання не просто на такий самий список, як змінна `a`, а на *той самий список*, що є змінна `a`. Таким чином, після виконання команди `b=a` і змінна `a`, і змінна `b` посилаються

на один і той же список. Тому, коли ми командою `b[1]=0` змінюємо елемент у списку, на який посилається змінна `b`, ми тим самим змінюємо той же список, на який посилається і змінна `a`.

Природно виникає запитання: що робити, якщо нам потрібно створити копію списку (тобто якщо нам потрібно, щоб дві змінні посилалися на різні списки з однаковими значеннями)? Є декілька варіантів вирішення проблеми. По-перше, можемо скористатися операцією отримання зрізу. Наприклад, значенням виразу `a[:]` є список, такий же, як той, на який посилається змінна `a`. Тому якщо змінній `b` значення присвоїти командою `b=a[:]`, то змінні `b` і `a` будуть посилятися на фізично різні списки, але значення в цих списках будуть однакові. Як наслідок, після виконання команди `b[1]=0` список `b` зміниться (буде `[10, 0, 30]`), а список `a` залишиться без змін (значення `[10, 20, 30]`).

Також для створення копії списку використовують метод `copy()`, який викликають зі списку, для якого створюється копія. Наприклад, у результаті виконання команди `b=a.copy()` створюється копія того списку, на який посилається змінна `a`, і посилання на цей новий список записується у змінну `b`. Тут ситуація така ж, як і під час використання зрізу для створення копії. У лістингу 4.4 наведено приклад програми, в якій різними способами створюють копії списків.

Лістинг 4.4. Створення копії списку

```
# Базовий список
a=[10,20,30]
# Перевіряємо вміст списку
print("Список a:",a)
# Присвоювання змінних
b=a
# Створення копії списку за допомогою зрізу
c=a[:]
# Створення копії списку за допомогою методу copy()
d=a.copy()
# Перевіряємо вміст списків
print("Список b:",b)
print("Список c:",c)
print("Список d:",d)
```

Розділ 4. Робота зі списками і кортежами

```
print("Змінюємо значення елемента a[1]=0.")  
# Змінюємо елемент у базовому списку  
a[1]=0  
# Перевіряємо вміст списків  
print("Список a:",a)  
print("Список b:",b)  
print("Список c:",c)  
print("Список d:",d)
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 4.4)

```
Список a: [10, 20, 30]  
Список b: [10, 20, 30]  
Список c: [10, 20, 30]  
Список d: [10, 20, 30]  
Змінюємо значення елемента a[1]=0.  
Список a: [10, 0, 30]  
Список b: [10, 0, 30]  
Список c: [10, 20, 30]  
Список d: [10, 20, 30]
```

На перший погляд може здатися, що питання зі створенням копії списку прояснилося. Але це тільки на перший погляд. Аби показати, що не все так просто, розглянемо ще один невеликий приклад. Нехай список $x=[10, 20, [100, 200], 30, [300, 400]]$. Важливо те, що серед елементів списку x наявні елементи-списки (це список $[100, 200]$, який є елементом $x[2]$, і список $[300, 400]$, який є елементом $x[4]$). Ми створимо командами $y=x[:]$ і $z=x.copy()$ дві копії списку x , змінимо командами $x[2][1]=0$ і $x[4]=0$ елементи цього списку і потім перевіримо, як ці зміни вплинули на копії y і z списку x .



Елемент $x[2]$ — це третій від початку елемент списку x . Легко помітити, що цей елемент сам є списком: це список $[100, 200]$, що складається з двох елементів. Для нас важливо те, що $x[2]$ — це список. Тому вираз $x[2][1]$ є посиланням на елемент із індексом 1 у списку $x[2]$. Легко зрозуміти, що

йдеться про другий елемент у списку [100, 200], тобто про елемент зі значенням 200. Тому командою $x[2][1]=0$ у списку [100, 200] (який сам є елементом списку x) значення 200 замінюється на значення 0.

Вираз $x[4]$ є посиланням на елемент [300, 400] списку x. Цей елемент також є списком. Командою $x[4]=0$ йому як значення замість посилання на список присвоюється число.

Відповідний програмний код наведено в лістингу 4.5.

Лістинг 4.5. Копіювання вкладених списків

```
# Вихідний список
x=[10,20,[100,200],30,[300,400]]
# Копія списку
y=x[:]
# Копія списку
z=x.copy()
# Перевіряємо вміст списків
print("Список x:",x)
print("Список y:",y)
print("Список z:",z)
print("Змінюємо елементи: x[2][1]=0 і x[4]=0.")
# Змінюємо значення елемента у внутрішньому списку
x[2][1]=0
# Змінюємо елемент у зовнішньому списку
x[4]=0
# Перевіряємо вміст списків
print("Список x:",x)
print("Список y:",y)
print("Список z:",z)
```

Виконання цього програмного коду дає такий результат:

Результат виконання програми (з лістингу 4.5)

```
Список x: [10, 20, [100, 200], 30, [300, 400]]
Список y: [10, 20, [100, 200], 30, [300, 400]]
Список z: [10, 20, [100, 200], 30, [300, 400]]
```

Змінюємо елементи: $x[2][1]=0$ і $x[4]=0$.

Список x : [10, 20, [100, 0], 30, 0]

Список y : [10, 20, [100, 0], 30, [300, 400]]

Список z : [10, 20, [100, 0], 30, [300, 400]]

Нагадаємо, що нас цікавить, наскільки зміни у вихідному списку x впливають на копії y і z . За логікою, зміни в списку x не повинні впливати на списки y і z . Що ж ми бачимо? По-перше, із самим списком x відбувається те, що й очікувалося: після присвоювання значень елементам списку, список змінюється. По-друге, зміна елемента $x[4]$ справді не відбивається на списках y і z . Тут немає нічого несподіваного. Але є сюрприз, пов'язаний із командою $x[2][1]=0$ (і це, по-третє). Річ у тому, що хоча зміни вносяться до списку x , вони мають місце і в списках y і z . Зрозуміло, що ситуація вимагає пояснень.

Доцільно детальніше зупинитися на тому, як створюється копія списку, в якому є елемент-список. Щоб легше було зрозуміти, виокремимо ключові моменти і терміни:

- є список, який копіюють, — це *вихідний*, або *копійований список*;
- у копійованому списку є елемент, який сам є списком, — це *елемент-список*;
- список, створений внаслідок копіювання вихідного, копійованого списку, — це *список-копія*.

У принципі, процес простий: для кожного з елементів створюється копія. Якщо елемент — число, то в списку-копії відповідний елемент має таке ж числове значення. Якщо у вихідному списку елемент — це список, то ситуація дещо інша. Такий елемент насправді *посилается* на список. Елемент-список у списку-копії буде мати таке ж значення, як і елемент-список у вихідному, копійованому списку. Але, грубо кажучи, це значення — адреса списку, на який посилається елемент. Або іншими словами, у списку-копії елемент-список технічно отримує як значення достоту таке ж посилання, як і аналогічний елемент у вихідному списку. Тому й виходить, що обидва елементи (у вихідному списку і списку-копії) посилаються фізично на один і той же список.

Phyton

Копії, які створюють за допомогою операції отримання зрізу або методу `copy()`, зазвичай називають *поверхневими копіями*. Якщо нам потрібно, щоб під час створення копії списку створювалися копії навіть для внутрішніх списків (тобто списків, що є елементами списків), є сенс скористатися функцією створення «глибокої», або *повної копії* `deepcopy()`. Як аргумент функції передається список, для якого створюють копію. Функція стає доступною після підключення модуля `copy`. Якщо інструкція підключення модуля має вигляд `import copy`, то функція викликається у форматі `copy.deepcopy()`. Приклад використання функції `deepcopy()` наведено в лістингу 4.6.

Лістинг 4.6. Створення повної копії списку

```
# Імпортуємо модуль copy
import copy
# Вихідний список із елементами-списками
A=[[10,20],[[30,40],[50,60]]]
# Повна копія списку
B=copy.deepcopy(A)
# Перевіряємо вміст списків
print("Список А:",A)
print("Список В:",B)
print("Виконуються команди A[0][1]=0 і A[1][1][1]=0.")
# Змінюємо елементи списку
A[0][1]=0
A[1][1][1]=0
# Перевіряємо вміст списків
print("Список А:",A)
print("Список В:",B)
```

У результаті виконання коду отримуємо таке:

Результат виконання програми (з лістингу 4.6)

```
Список А: [[10, 20], [[30, 40], [50, 60]]]
Список В: [[10, 20], [[30, 40], [50, 60]]]
Виконуються команди A[0][1]=0 і A[1][1][1]=0.
Список А: [[10, 0], [[30, 40], [50, 0]]]
Список В: [[10, 20], [[30, 40], [50, 60]]]
```

Розділ 4. Робота зі списками і кортежами

Вихідний список створено за допомогою команди `A=[[10,20], [[30,40], [50,60]]]`. У цьому списку два елементи. Перший елемент `[10,20]` (посилання `A[0]`) є списком із двох числових елементів. Другий елемент `[[30,40], [50,60]]` (посилання `A[1]`) — це теж список із двох елементів, але ці елементи, у свою чергу, також є списками (посилання `A[1][0]` на список `[30,40]` і посилання `A[1][1]` на список `[50,60]`). Таким чином, маемо справу з ієрархією вкладених списків. Командою `B=copy.deepcopy(A)` створюємо повну копію списку `A`. Таким чином, змінні `A` і `B` посилаються на фізично різні списки, але з однаковими елементами. Тому відображаючи за допомогою функції `print()` вміст списків `A` і `B`, отримуємо однакові результати.

На наступному етапі командами `A[0][1]=0` і `A[1][1][1]=0` змінюються значення елементів у внутрішніх списках. Після цього знову перевіряємо вміст списків `A` і `B`. Легко помітити, що список `A` змінився, а список `B` — не змінився.



Є одна цікава особливість оператора присвоювання, яка має відношення до списків і про яку бажано знати. Стосується вона **множинного присвоювання**, коли в лівій частині від оператора присвоювання вказується (через кому) декілька змінних, а праворуч (теж через кому) — значення, які присвоюються змінним. Наприклад, у результаті виконання команди `x, y, z=1, 2, 3` змінний `x` буде присвоєно значення 1, змінний `y` — значення 2, а змінний `z` — значення 3. Взагалі кількість змінних ліворуч від оператора присвоювання повинна збігатися з кількістю значень праворуч від оператора присвоювання (щоб була однозначна відповідність). Але може бути й більш заплутана ситуація, коли праворуч значень більше, ніж змінних ліворуч. Тоді, якщо перед однією зі змінних поставити зірочку `*`, то всі «зайві» значення будуть записані в цю змінну у вигляді списку. Наприклад, під час виконання команди `x, *y, z=1, 2, 3, 4, 5` змінна `x` отримає значення 1, змінна `z` — значення 5, а змінна `y` буде посиланням на список `[2, 3, 4]`.

Ще одне зауваження стосується **багатократного присвоювання** (у виразі — декілька операторів присвоювання). Наприклад, у результаті виконання команди `x=y=1` змінні `x` і `y` отримують значення 1. Причому, якщо потім змінити значення однієї зі змінних (наприклад, виконати команду `x=2`), то значення іншої змінної (у цьому разі `y`) не зміниться. Але це у випадку, якщо не йдеться про списки. Скажімо, виконується команда `x=y=[1, 2]`. Після її виконання змінні `x` і `y` посилаються не просто на однакові списки,

а на один і той самий список `[1, 2]`. Якщо ми присвоїмо змінній `x` інший список (або значення іншого типу), змінна `y` не зміниться. Так, якщо після команди `x=y=[1, 2]` виконується команда `x=[3, 4]`, то змінна `x` буде посилятися на список `[3, 4]`, а змінна `y` буде посилятися на вихідний список `[1, 2]`. Але якщо ми змінимо тільки елемент у вихідному списку, то зміниться значення обох змінних. Припустімо, після команди `x=y=[1, 2]` виконується команда `x[0]=0`. Після цього змінна `x` посилається на список `[0, 2]` (як і повинно бути), але і змінна `y` посилається на цей список! Щоб зрозуміти, чому так відбувається, необхідно знову ж таки врахувати, що в Python змінні посилаються на значення. Тому важливо проводити чітку межу між зміною посилання і зміною значення, на яке зроблено посилання.

Списки та функції

У мене є свої думки, сильні думки, але я не завжди погоджуся з ними.

Дж. Буш (молодший)

Враховуючи особливу роль списків у мові Python, далі розглянемо деякі моменти (в основному, прикладного характеру), пов'язані з використанням списків. У цьому пункті зупинимося на тому, як концепція списків співвідноситься з концепцією функцій. Іншими словами, у сферу нашого інтересу на цьому етапі потраплять питання спільнотого використання функцій і списків.

Одразу ж в око впадають дві очевидні ситуації: це передача списку аргументом функції і повернення функцією списку як результату. З формальної точки зору, тут немає нічого складного. Так, якщо список передають аргументом функції, то під час обробки такого аргументу просто необхідно врахувати, що маемо справу саме зі списком. Якщо функція повертає як результат список, то в тілі функції список повинен бути сформований, а потім за допомогою інструкції `return` «виданий» як значення функції (повертається насправді посилання на список).

Як ілюстрацію розглянемо програмний код, у якому визначено і потім використано декілька функцій. Зокрема, описано функцію `rand_vector()` для створення списку зазначененої довжини (визначається аргументом функції) із випадковими ціличисловими елементами, функцію `show()` для відображення вмісту списку, а також функцію `scal_prod()` для обчислення скалярного добутку векторів.



Якщо є два вектори $\vec{a} = (a_1, a_2, \dots, a_n)$ і $\vec{b} = (b_1, b_2, \dots, b_n)$, то скалярним добутком цих векторів називають число $\vec{a} \cdot \vec{b} = a_1b_1 + a_2b_2 + \dots + a_nb_n$.

Програмний код, у якому реалізовано перераховані вище функції, а також проілюстровано їхні можливості, наведено в лістингу 4.7.



Лістинг 4.7. Функції та списки

```
# Підключаємо модуль для
# генерування випадкових чисел
import random

# Функція для відображення вмісту списку.
# Аргумент функції - список, який відображується
def show(a):
    # Довжина списку
    n=len(a)
    # Початкова фраза
    print(n,"D-вектор: <",sep="",end="")
    # Оператор циклу для відображення елементів
    for i in range(n-1):
        # Відображається елемент списку
        print(a[i],end=" | ")
    # Відображається останній елемент списку
    print(a[n-1],">>.",sep="")
# Функція для створення списку з випадковими
# цілочисловими елементами.
# Аргумент функції - кількість елементів у списку
def rand_vector(n):
    # Створюємо порожній список
    r=[]
    # Оператор циклу для додавання нових
    # елементів у кінець списку
    for i in range(n):
        # Додаємо в кінець списку новий
        # елемент - випадкове ціле число
        # в діапазоні від 0 до 6
        r.append(random.randint(0,6))
```

Розділ 4. Робота зі списками і кортежами

```
# Значення функції - список
return r

# Функція для обчислення скалярного добутку
# двох векторів. Аргументи функції - списки, на основі
# яких реалізуються вектори
def scal_prod(a,b):
    # Довжина першого списку
    Na=len(a)
    # Довжина другого списку
    Nb=len(b)
    # Найменше з двох значень
    N=min(Na,Nb)
    # Початкове значення суми (визначається
    # через скалярний добуток)
    s=0
    # Оператор циклу для обчислення суми
    for i in range(N):
        # Додаємо до суми новий доданок
        s+=a[i]*b[i]
    # Результат функції
    return s

# Ініціалізація генератора випадкових чисел
random.seed(2014)
# Перший випадковий вектор
a=rand_vector(3)
# Другий випадковий вектор
b=rand_vector(5)
# Відображаємо перший вектор
show(a)
# Відображаємо другий вектор
show(b)
# Обчислюємо скалярний добуток
p=scal_prod(a,b)
# Відображаємо результат для скалярного добутку
print("Скалярний добуток:",p)
```

Результат виконання програмного коду такий:

■ Результат виконання програми (з лістингу 4.7)

3D-вектор: $\langle 2 | 4 | 3 \rangle$.

5D-вектор: $\langle 1 | 2 | 1 | 3 | 3 \rangle$.

Скалярний добуток: 13

Оскільки в програмному коді ми збираємося генерувати випадкові числа, командою `import random` підключаємо модуль `random`, призначений саме для таких цілей.



У цьому випадку нам знадобляться всього дві функції з модуля `random`, хоча насправді там дуже багато інших корисних функцій.

При відображені вмісту списку явно зазначається кількість елементів у списку, а самі елементи розміщують у кутових дужках із вертикальною рискою, що відіграє роль роздільника. Усе це реалізовано у функції `show()`, призначений для відображення вмісту списку (переданого аргументом функції — аргумент позначене як `a`). У тілі функції за допомогою команди `n=len(a)` обчислюють довжину списку (кількість елементів у списку). Після цього командою `print(n, "D-вектор: <", sep="", end="")` відображається текст. Перші два аргументи функції `print()` — це кількість елементів у списку `n` і текст "D-вектор: <". Функції `print()` за ключем передано аргументи `sep=""` і `end=""` (обидва значення — порожні текстові рядки). Такі аргументи означають буквально таке: роздільника (аргумент `sep`) між відображуваними елементами немає (точніше, роздільник — порожній текст), а в кінці (аргумент `end`) перехід до нового рядка не виконується (замість використовуваної за замовчуванням інструкції переходу до нового рядка — порожній текст). Для відображення елементів списку `a` в операторі циклу індексна змінна `i` пробігає значення від 0 до `n-2` (нагадаємо, що інструкція вигляду `range(m)` повертає віртуальну послідовність цілих чисел від 0 до `m-1` включно) — тобто перебираються всі елементи списку, крім останнього. Командою `print(a[i], end=" | ")` відображається черговий елемент списку, а одразу після цього елемента — вертикальна риска. Перехід до нового рядка не відбувається. Після закінчення виконання оператора циклу командою `print(a[n-1], ">.", sep="")` відображається останній елемент

і закриваюча кутова дужка. Роздільником між цими текстовими фрагментами є порожній текст (завдяки переданому за ключем аргументу `sep=""`).

Функція для створення списку з випадковими ціличисловими елементами називається `rand_vector()`. У неї один аргумент (позначено як `n`) — передбачається, що це ціле число, яке визначає розмір списку. У тілі функції командою `x = []` створюється порожній список. Після цього запускається оператор циклу, і за кожен цикл командою `x.append(random.randint(0, 6))` у список (кінець списку) додається новий елемент — випадкове ціле число в діапазоні від 0 до 6. Для генерування випадкових чисел ми використали функцію `randint()` із модуля `random`. Як аргументи функції передаються межі діапазону, в якому генеруються цілі числа. Результат функції `randint()`, як нескладно згадатися, — ціле число.



Як зазначалося вище, у модулі `random` багато інших функцій, використовуваних, крім іншого, для генерування випадкових чисел. Наприклад:

- функція `random()` генерує випадкове дійсне число в діапазоні від 0 (включно) до 1 (не включаючи);
- функція `uniform()` генерує рівномірно розподілені дійсні числа у певному діапазоні — межі діапазону передаються як аргументи функції;
- функція `betavariate()` повертає випадкове число, що має бета-розподіл;
- функція `expovariate()` повертає випадкове число, що має експоненціальний розподіл;
- функція `gammavariate()` повертає випадкове число, що має гамма-розподіл;
- функція `gauss()` повертає випадкове число, що має розподіл Гаусса;
- функція `lognormvariate()` повертає випадкове число, що має логнормальний розподіл;
- функція `normalvariate()` повертає випадкове число, що має нормальній розподіл (фактично те ж саме, що й розподіл Гаусса);
- функція `paretovariate()` повертає випадкове число, що має розподіл Парето;
- функція `weibullvariate()` повертає випадкове число, що має розподіл Вейбулла.

Як аргументи цим функціям передаються параметри відповідних розподілів (якщо такі є).

Для додавання нового елемента до списку використано метод `append()`, який викликається зі списку. Після того, як список `r` сформовано, командою `return r` він повертається результатом функції `rand_vector()`.

За допомогою функції `scal_prod()` обчислюється скалярний добуток двох векторів, які ми ототожнюємо зі списками. Аргументи функції — списки. Результат функції — це сума попарних добутків елементів списків. Тут ми зустрічаемося з потенційною проблемою: у списках може бути різна кількість елементів. У такому разі неявно будемо вважати, що елементи, яких не вистачає, нульові. А оскільки добуток нуля на будь-яке число дорівнює нулю, і відповідний доданок у загальну суму внеску не робить, то замість того щоб у «короткому» списку дописувати нульові елементи, ми просто можемо ігнорувати зайві елементи в «довшому» списку. Саме такий підхід використано при визначенні результату функції `scal_prod()`. У тілі функції командами `Na=len(a)` і `Nb=len(b)` визначається довжина (кількість елементів) першого і другого списків, переданих аргументами функції. Потім командою `N=min(Na, Nb)` у змінну `N` записується мінімальне з двох значень `Na` і `Nb`. Також командою `s=0` задаємо нульове початкове значення для змінної, в яку буде записуватися сума — результат скалярного добутку. В операторі циклу індексна змінна `i` «пробігає» значення від 0 до `N-1` і служить індексом для перебору елементів у першому і другому списках. За кожну ітерацію командою `s+=a[i]*b[i]` значення змінної `s` збільшується на величину добутку відповідних елементів (елементів із однаковими індексами) списків `a` і `b`. Після закінчення оператора циклу командою `return s` значення змінної `s` повертається як результат функції.

Окрім описаних функцій, у програмному коді є такі команди:

- Інструкція `random.seed(2014)` виконує ініціалізацію генератора випадкових чисел. Узагалі, ця команда необов'язкова. Можна було її не використовувати зовсім. Якщо ми все ж викликаємо функцію `seed()` із деяким числовим аргументом, то генератор під час кожного запуску програми буде генерувати одну й ту ж послідовність випадкових чисел.



Пікантність ситуації в тому, що будь-який генератор випадкових чисел насправді «видає» зовсім не випадкові числа. Тому правильніше говорити про

Розділ 4. Робота зі списками і кортежами

псевдовипадкові числа. Зовнішньо дуже схоже, що числа випадкові — але це не так. Для отримання послідовності таких чисел використовують певну формулу або алгоритм. І щоб усе це «запустити», потрібно якесь початкове значення, яке беруть за основу при обчисленні послідовності псевдовипадкових чисел. Таке вкрай важливе для генератора випадкових чисел значення зазвичай задається під час ініціалізації генератора випадкових чисел — фактично, у цьому ініціалізації генератора чисел і полягає. Якщо ми в явному вигляді ініціалізуємо генератор, викликавши функцію `seed()` із певним аргументом, то цей аргумент використовується для «старту» обчислювально-го процесу. Тому кожного разу послідовність псевдовипадкових чисел буде одна й та ж. Якщо явно ініціалізація генератора випадкових чисел не виконується, то початкове значення визначається на основі системного часу, тому кожного разу під час запуску програми отримуємо нову послідовність псевдовипадкових чисел.

- Командами `a=rand_vector(3)` і `b=rand_vector(5)` створюються два випадкових списки з кількістю елементів 3 і 5 відповідно.
- Командами `show(a)` і `show(b)` уміст списків відображаємо в консолі.
- Командою `r=scal_prod(a,b)` обчислюємо скалярний добуток, а командою `print("Скалярний добуток:",r)` відображаємо результат обчислення скалярного добутку.

Вкладені списки

Я у відповіді за всі мої помилки. І ви — також.

Дж. Буш (молодший)

Хоча різних варіантів списків дуже багато, на одному типі списків зу-пинимося докладніше. Це *вкладені списки* — списки, елементами яких, у свою чергу, теж є списки. Нас цікавитиме прикладний аспект у використанні такого роду «конструкцій», а саме: як створювати вкладені списки і як їх потім використовувати на практиці. Зрозуміло, що ці задачі можна вирішувати різними способами. Розгляньмо невеликий приклад, у якому описано декілька функцій, призначених для роботи з матрицями: є функція для створення матриці з елементами — випадковими числами, є функція для створення одиничної матриці, описано функцію для обчислення добутку квадратних матриць, а також є функція для відображення (по рядках) матриці у вікні виводу (консолі).



Незайвим буде освіжити в пам'яті деякі моменти, пов'язані з матрицями. Матрицею називається набір чисел, упорядкованих за рядками і стовпчиками — тобто у вигляді таблиці. Квадратною називається матриця, в якої одна-кова кількість рядків і стовпчиків. Матриця, таким чином, описується набо-ром елементів: якщо про матрицю \hat{A} говорять, що вона складається з елементів a_{ij} (індекси $i = 1, 2, \dots, n$ і $j = 1, 2, \dots, m$), то це означає, що на перетині i -го рядка і j -го стовпчика знаходиться число a_{ij} .

Однійна матриця — це така квадратна матриця, в якої на головній діагона-лі одиничні елементи, а всі інші елементи дорівнюють нулю: тобто $a_{ii} = 1$, якщо $i = j$, і $a_{ij} = 0$, якщо $i \neq j$.

Розділ 4. Робота зі списками і кортежами

Якщо є дві квадратні матриці \hat{A} з елементами a_{ij} і \hat{B} з елементами b_{ij} (індекси $i, j = 1, 2, \dots, n$), то добутком цих матриць називається матриця $\hat{C} = \hat{A}\hat{B}$ з елементами $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}$. Слід зауважити, що

множити можна не тільки квадратні матриці. Але в будь-якому разі кількість стовпчиків першої матриці повинна збігатися з кількістю рядків другої матриці.

Відповідний програмний код наведено в лістингу 4.8.

Лістинг 4.8. Використання вкладених списків

```
# Підключаємо модуль для генерування
# випадкових чисел
from random import *
# Функція для створення матриці
# з випадковими елементами
def rand_matrix(n,m):
    # Створюємо матрицю з елементами - випадковими числами
    A=[[randint(0,9) for j in range(m)] for i in range(n)]
    # Результат функції
    return A
# Функція для створення одиничної матриці
def unit_matrix(n):
    # Одинична матриця
    A=[[int(i==j) for i in range(n)] for j in range(n)]
    # Результат функції
    return A
# Функція для обчислення добутку квадратних матриць
def mult_matrix(A,B):
    # Розмір матриці
    n=len(A)
    # Матриця-результат із початковими нульовими значеннями
    C=[[0 for i in range(n)] for j in range(n)]
    # Перший індекс
    for i in range(n):
        # Другий індекс
        for j in range(n):
```

Phyton

```
# Внутрішній індекс, за яким береться сума
for k in range(n):
    # Додаємо доданок у суму
    C[i][j]+=A[i][k]*B[k][j]
# Результат функції
return C

# Функція для відображення матриці
def show_matrix(A):
    # Перебирання рядків матриці
    for a in A:
        # Перебирання елементів у рядку матриці
        for b in a:
            # Відображаємо елемент матриці
            print(b,end=" ")
        # Перехід до нового рядка
        # (під час виводу в консоль)
        print()

# Ініціалізація генератора випадкових чисел
seed(2014)

# Матриця з випадковими числами
A=rand_matrix(3,5);
# Матриця у вигляді списку
print("Список:",A)
# Матриця як вона є
print("Ця ж матриця:")
show_matrix(A)
# Одинична матриця
E=unit_matrix(4)
# Відображаємо матрицю
print("Одинична матриця:")
show_matrix(E)
# Перша матриця
A1=rand_matrix(3,3)
# Друга матриця
A2=rand_matrix(3,3)
# Добуток матриць
A3=mult_matrix(A1,A2)
```

```
# Відображаємо першу матрицю
print("Перша матриця:")
show_matrix(A1)
# Відображаємо другу матрицю
print("Друга матриця:")
show_matrix(A2)
# Відображаємо добуток матриць
print("Добуток матриць:")
show_matrix(A3)
```

У програмному коді за допомогою команди `from random import *` підключаемо модуль `random` для генерування випадкових чисел (причому за такого способу підключення для виклику функцій із цього модуля його ім'я явно вказувати не треба). Нам знадобиться функція `randint()` із модуля `random`, за допомогою якої будемо генерувати елементи матриці в тілі функції `rand_matrix()`. Функцію описано з двома аргументами `n` і `m`. Вони визначають відповідно кількість рядків і стовпчиків у матриці, яка повертається як результат. Матрицю створюємо за допомогою команди `A=[[randint(0, 9) for j in range(m)] for i in range(n)]`. У цьому разі використовуємо вкладені *генератори списків*. Інструкція `[randint(0, 9) for j in range(m)]` формує список із `m` елементів, причому кожний елемент — результат виклику функції `randint(0, 9)`, тобто ціле випадкове число в діапазоні від 0 до 9 включно. Саму ж інструкцію `[randint(0, 9) for j in range(m)]` указано як таку, що формує елемент списку для зовнішнього генератора списків. У цьому зовнішньому генераторі списків індексна змінна «пробігає» ціличислові значення від 0 до `n-1`. Тому в результаті отримуємо список із `n` елементів, причому кожний елемент цього списку сам є списком із `m` елементів (а ці елементи — випадкові числа).

Функція для створення одиничної матриці називається `unit_matrix()`, і в ній один аргумент (позначено як `n`) — розмір (ранг) матриці (кількість рядків і вона ж — кількість стовпчиків, оскільки матриця, за визначенням, квадратна). Одиничну матрицю `A` у тілі функції створюємо командою `A=[[int(i==j) for i in range(n)] for j in range(n)]`. Команда дуже схожа на ту, що була в тілі функції `rand_matrix()`, але є невеликі відмінності. По-перше, обидві індексні змінні (у зовнішньому і внутрішньому генераторах списків) «пробігають» однакові діапазони значень

(у квадратній матриці кількість рядків дорівнює кількості стовпчиків). По-друге, під час формування елементів у внутрішньому генераторі списків замість функції `randint()` використано інструкцію `int(i==j)`. Тут логіка наступна: значення виразу `i==j` дорівнює `True`, якщо індекси `i` і `j` однакові, `i False`, якщо ці індекси різні. Логічне значення при перетворенні за допомогою функції `int()` у цілочисловий вигляд дає 1 там, де було `True`, і 0 там, де було `False`. У результаті отримуємо список, що складається з `n` списків, кожний із яких, у свою чергу, складається з `n` нулів або одиниць. У кожному внутрішньому списку тільки одна одиниця (інші елементи нульові), причому одиниця у внутрішньому списку знаходиться на тій позиції, яка збігається з позицією внутрішнього списку в зовнішньому списку. Мовою матриць це означає, що одиниці знаходяться тільки на головній діагоналі — тобто ми створили одиничну матрицю. Ця матриця є результатом функції `unit_matrix()`.

Функція `mult_matrix()` призначена для обчислення добутку квадратних матриць. Ми вважаємо, що її аргументи `A` і `B` є саме тими матрицями, які потрібно перемножити. Результат функції — матриця, яку отримуємо, перемноживши матриці, передані аргументами функції.

У тілі функції командою `n=len(A)` визначаємо розмір матриці — первого аргументу функції `mult_matrix()`. Строго кажучи, у змінну `n` записується кількість елементів у списку `A`. Елементами списку `A`, у свою чергу, також є списки (внутрішні списки). Кількість елементів у списку `A` — це кількість рядків у відповідній матриці. Кількість елементів у внутрішніх списках — кількість стовпчиків матриці. Ми виходимо з того, що матриця `A` квадратна, і кількість стовпчиків у неї така ж, як кількість рядків. Більше того, ми також припускаємо, що другий аргумент функції `mult_matrix()` — матриця `B`, — теж квадратна і такого ж розміру, що й матриця `A`. Але перевірка всіх цих важливих моментів у програмному коді не реалізується. Хто бажає, нехай подумає, як її доцільно було б реалізувати.



Зручний механізм обробки можливих «неприємностей» (помилок, що виникають під час виконання програми) пов'язаний з використанням конструкції `try-except`. Цей підхід ми вже обговорювали в одному з попередніх розділів.

Розділ 4. Робота зі списками і кортежами

У тілі функції `mult_matrix()` за допомогою команди `C=[[0 for i in range(n)] for j in range(n)]` створюємо вкладений список `C` із нульовими елементами. Даний список задає структуру матриці-результату функції. Усі елементи цієї матриці перед початком обчисень нульові.

Далі в тілі функції запускаються три вкладені оператори циклу: у перших двох операторах перебираються індекси елементів матриці `C`, а за індексом третього оператора циклу обчислюється сума. При цьому виконується лише одна команда `C[i][j] += A[i][k] * B[k][j]`, якою сума для значення елемента `C[i][j]` збільшується на величину добутку елементів `A[i][k]` і `B[k][j]` матриць `A` і `B`. Після закінчення обчисень матриця `C` повертається як результат функції.

Для відображення матриць у більш прийнятному, з математичної точки зору, вигляді визначаємо функцію `show_matrix()`, аргументом якої передається матриця для відображення.



Якщо спробувати відобразити вкладений список за допомогою функції `print()`, то елементи списку ми, звичайно ж, побачимо. Але відображатися вони будуть не у вигляді таблиці, як прийнято відображати матриці, а як вкладені списки.

У тілі функції `show_matrix()` запускаються вкладені оператори циклу. У зовнішньому циклі змінна `a` «пробігає» набір значень зі списку `A`. Іншими словами, змінна `a` послідовно набуває значення елементів зі списку `A`. Тому змінну `a` можемо інтерпретувати як елемент списку `A`. Оскільки список `A` сам складається зі списків, то `a` — це список. Там знаходяться елементи, які відповідають певному ряду в матриці `A`. У другому, внутрішньому операторі циклу змінна `b` «пробігає» значення елементів зі списку `a`. Таким чином, якщо `a` — це ряд елементів у матриці `A`, то `b` — це один із елементів цього ряду. Команда `print(b, end=" ")` відображає елемент ряду у вікні виводу, перехід до нового рядка не відбувається, а замість цього додається пробіл. Таким чином, елементи ряду відображаються в один рядок через пробіл. Після того, як елементи певного ряду матриці перебрано і роботу внутрішнього оператора циклу закінчено, виконується команда `print()`. У результаті виконується перехід до нового

рядка для виводу інформації. У зовнішньому операторі циклу змінна `a` набуває нового значення, й усе починається знову: елементи ряду відображаються в один рядок через пробіл і так далі.

Окрім описаних функцій є декілька команд, які дозволяють проілюструвати, як ці самі функції можна використовувати.

Ініціалізацію генератора випадкових чисел виконуємо командою `seed(2014)`. Ще раз нагадуємо, що таку ініціалізацію можна було бінеробити.

Матрицю з випадковими числами створюємо командою `A=rand_matrix(3,5)`. У цьому разі йдеться про матрицю з 3 рядків і 5 стовпчиків. Для порівняння командою `print("Список:", A)` відображаємо вміст списку `A` стандартним способом, а потім командою `show_matrix(A)`.

Однічну матрицю рангу 4 створюємо за допомогою команди `E=unit_matrix(4)`. Матрицю відображаємо командою `show_matrix(E)`.

Крім цього, командами `A1=rand_matrix(3,3)` і `A2=rand_matrix(3,3)` створюємо дві випадкові квадратні матриці рангу 3. Добуток цих матриць обчислюємо за допомогою інструкції `A3=mult_matrix(A1,A2)`. Після цього командами `show_matrix(A1)`, `show_matrix(A2)` і `show_matrix(A3)` відображаємо дві вихідні матриці, а також матрицю-добуток.

Нижче показано результат виконання програмного коду:



Результат виконання програми (з лістингу 4.8)

```
Список: [[5, 9, 6, 3, 5], [3, 6, 6, 7, 0], [9, 6, 5, 2, 9]]
```

Ця ж матриця:

```
5 9 6 3 5
3 6 6 7 0
9 6 5 2 9
```

Однійна матриця:

```
1 0 0 0  
0 1 0 0  
0 0 1 0  
0 0 0 1
```

Перша матриця:

```
7 2 7  
3 4 0  
1 2 3
```

Друга матриця:

```
3 4 6  
6 6 5  
1 2 4
```

Добуток матриць:

```
40 54 80  
33 36 38  
18 22 28
```

Хто бажає, може самостійно перевірити правильність обчислення добутку матриць.

Одна з конкурентних переваг мови Python полягає якраз у тому, що одна й та сама задача може розв'язуватися різними способами, до того ж не-рідко ці способи разюче відрізняються. Тому очевидно, що запропонований вище підхід не є єдино можливим. Із деякими іншими концепціями створення програмних кодів ми ще познайомимося в ході вивчення матеріалу книги.



При створенні списків (у тому числі й вкладених) нерідко використовують оператор повторення * (зірочка). Якщо цей бінарний оператор (який формально збігається з оператором множення) застосувати до списку (один операнд — список, інший операнд — ціле число), то результатом буде новий список,

який утворюється внаслідок повторення елементів із вихідного списку. Кількість повторів визначається цілочисловим операндом. Наприклад, у результаті виконання команди $A = [0]^*5$ (або команди $A = 5 * [0]$) змінна A буде посилатися на список із п'яти нулів $[0, 0, 0, 0, 0]$. Якщо $B = [1, 2]^*3$, то насправді результатом буде список, утворений унаслідок трикратного повторення пари елементів 1 і 2, тобто список $[1, 2, 1, 2, 1, 2]$. Або, скажімо, після виконання команди $C = [[1]^*2]^*3$ змінна C буде посилатися на список $[[1, 1], [1, 1], [1, 1]]$. Чому так? Тому що результатом виразу $[1]^*2$ є список $[1, 1]$, а результатом виразу $[[1, 1]]^*3$ є список $[[1, 1], [1, 1], [1, 1]]$ (трикратне повторення елемента $[1, 1]$).

Варто зазначити, що оператор повторення * можна застосовувати не тільки до списків, а й, наприклад, до тексту.

Знайомство з кортежами

Сюди прямують катери та гелікоптери. Однак їм потрібен певний час, щоб доплисти.

Дж. Буш (молодший)

Кортеж являє собою впорядкований набір деяких елементів. У цьому сенсі він мало чим відрізняється від списку. Тобто і список, і кортеж — це впорядкований набір елементів. У чому ж відмінність? Відмінність — у способі або механізмі реалізації, а також у тому, які операції припустильні зі списками й кортежами. Можна також сказати, що в основі списків і кортежів одна й та ж структура — упорядкований набір елементів. Ця структура може реалізуватися через списки, а може — через кортежі.

Якщо від абстрактних питань перейти до більш практичних, то слід відзначити, що принципова відмінність кортежів від списків полягає в тому, що кортежі не можна змінювати. Тобто після того, як кортеж створено, внести до нього зміни вже не вийде.



На перший погляд, ця особливість кортежів дуже сильно обмежує сферу їхнього застосування. Але це тільки на перший погляд. Насправді, ця особливість швидше «технічного» характеру, і її у багатьох випадках удається уникнути. Наприклад, якщо змінна посилається на кортеж, і нам потрібно внести в кортеж зміни, то можна створити новий кортеж і посилання на цей кортеж присвоїти змінній.

З точки зору «типології» мови Python, кортеж належить до типу `tuple`. Отже, немає нічого дивного в тому, що створити кортеж можна за

допомогою функції `tuple()`. Якщо аргументу у функції немає, то буде створено порожній кортеж. Якщо аргументом функції `tuple()` передати, наприклад, список або текстовий рядок, отримаємо кортеж, що складається відповідно з елементів списку або букв текстового рядка. Ще ми можемо створити кортеж, якщо в круглих дужках перерахувати через кому елементи, які його формують.



Порожні круглі дужки означають порожній кортеж. Якщо в кортежі один елемент і ми створюємо такий кортеж за допомогою круглих дужок, то після цього елемента слід поставити кому. Якщо кому не поставити, то отримаємо не кортеж, а просто той елемент, який у круглих дужках. Наприклад, команда `a = ()` створює порожній кортеж, і посилання на нього записується в змінну `a`. Команда `a = (10, 20)` створює кортеж із двох елементів (10 і 20). А команда `a = (100)` кортеж не створює: змінна `a` посилається на ціличислове значення 100. Щоб створити кортеж, який складається всього з одного елемента 100, використовуємо команду `a = (100,)`.

Ще один важливий момент: якщо не використовувати круглі дужки, а просто перерахувати через кому декілька елементів (і присвоїти цю конструкцію змінній), отримаємо кортеж.

Звертання до елементів кортежу виконується так само, як і до елементів списку — за допомогою індексу. Індекс указується у квадратних дужках після імені кортежу. Індексація елементів кортежу починається з нуля. Так само як і для списку, для кортежу можна отримати зріз.



Ще раз нагадаємо, що отримання доступу до елемента кортежу або отримання зрізу для кортежу означає, що ми можемо «прочитати» відповідні значення, але змінити їх не можемо.

Деякі нескладні приклади створення й використання кортежів наведено в лістингу 4.9.



Лістинг 4.9. Знайомство з кортежами

```
# Створюємо порожній кортеж  
a=tuple()
```

```

# Перевіряємо вміст кортежу
print(a)
# Створюємо кортеж на основі списку
b=tuple([10,20,30])
# Перевіряємо вміст кортежу
print(b)
# Створюємо кортеж на основі тексту
c=tuple("Python")
# Перевіряємо вміст кортежу
print(c)
# Об'єднання кортежів
a=b+(40,[1,2,3],60)
# Перевіряємо результат об'єднання кортежів
print(a)
# Отримання зразу
print(a[2:5])

```

Виконавши цей програмний код, отримаємо такий результат:

Результат виконання програми (з лістингу 4.9)

```

()
(10, 20, 30)
('P', 'y', 't', 'h', 'o', 'n')
(10, 20, 30, 40, [1, 2, 3], 60)
(30, 40, [1, 2, 3])

```

Наприклад, командою `a=tuple()` створюється порожній кортеж, і посилання на нього записується в змінну `a`. Командою `b=tuple([10,20,30])` створюємо кортеж із трьох числових елементів. Приклад створення кортежу на основі текстового значення дає команда `c=tuple("Python")`.

Команда `a=b+(40,[1,2,3],60)` ілюструє одразу декілька «моментів», пов'язаних із використанням кортежів. По-перше, тут об'єднуються два кортежі. Один визначається змінною `b`, а другий кортеж `(40,[1,2,3],60)` задано явно. Для об'єднання кортежів використано оператор додавання. По-друге, у кортежі `(40,[1,2,3],60)` один із елементів — список `[1,2,3]`. По-третє, посилання на результат операції

`b+(40, [1, 2, 3], 60)` записується в змінну `a`, яка до цього посилається на порожній кортеж. Тут потрібно врахувати, що в результаті обчислення виразу `b+(40, [1, 2, 3], 60)` повертається новий кортеж, який утворюється внаслідок об'єднання кортежів `b` і `(40, [1, 2, 3], 60)`. Саме на цей новий кортеж буде посилятися змінна `a`. В цьому нескладно переконатися, наприклад, за результатом отримання зрізу для кортежу, на який посилається змінна `a`. А саме, щоб отримати зріз ми використовуємо інструкцію `a[2:5]` — тобто діємо так само, як і в разі отримання зрізу для списку.



Зверніть увагу: з різом кортежу є кортеж.

Методів у кортежів усього два. За допомогою методу `index()` за значенням елемента в кортежі можна дізнатися індекс цього елемента. Якщо елементів із таким значенням у кортежі декілька, повертається індекс першого елемента. За допомогою методу `count()` виконується підрахунок кількості елементів у кортежі з певним значенням (значення передається аргументом методу).

Дізнатися загальну кількість елементів у кортежі можна за допомогою функції `len()`. Як аргумент функції передається кортеж (або змінна, яка посилається на кортеж). Оператор `in` дозволяє визначити, чи входить елемент у кортеж.

Резюме

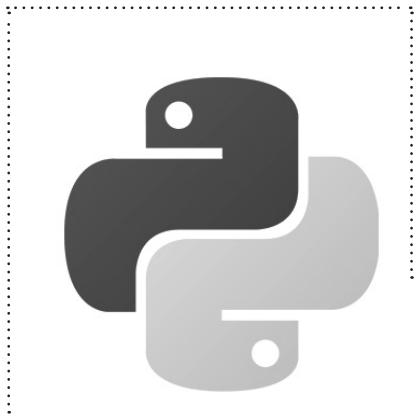
Я розумію, що кожен в цій країні не згоден із рішеннями, які я прийняв.

Дж. Буш (молодший)

- Список — це впорядкований набір елементів. Для створення списку елементи беруть у квадратні дужки. Також можна використовувати функцію `list()`.
- Елементи списку можуть бути різного типу.
- Визначити кількість елементів у списку можна за допомогою функції `len()`.
- Для отримання доступу до елемента списку після імені списку у квадратних дужках указують індекс елемента. Індексація елементів починається з нуля. Для індексації елементів із кінця списку використовують від'ємні індекси.
- Для отримання зрізу (доступу до декількох елементів) у квадратних дужках після імені масиву вказуються через двокрапку два індекси, які визначають послідовність елементів зрізу.
- Значення елементів у списку можна змінювати. Елементи зі списку можна видаляти (наприклад, методами `pop()` і `remove()`), додавати до списку (наприклад, за допомогою методів `append()` і `insert()`), а також замінювати одну групу елементів на іншу (скажімо, через операцію присвоювання значення зізгу). Списки можна об'єднувати (метод `extend()`).
- При копіюванні списків присвоювання зазвичай не використовується (оскільки при цьому копія списку не створюється,

просто дві змінні будуть посилатися на один список). Поверхневе копіювання списків здійснюється через отримання зрізу або за допомогою методу `copy()`. Під час створення поверхневої копії для внутрішніх списків копії не створюються. Копіювання списків можна виконувати за допомогою функції `deepcopy()` (із модуля `copy`). В останньому випадку створюється повна копія списку (тобто створюються копії навіть для внутрішніх списків).

- Кортеж, як і список, є впорядкованим набором елементів. Багато в чому кортежі схожі на списки, але кортежі не можна змінювати.
- Для створення кортежу використовують функцію `tuple()` або в круглих дужках через кому перераховують елементи кортежу.
- Доступ до елементів кортежу отримують за індексом. Індексація елементів кортежу починається з нуля. Індекс указують у квадратних дужках після імені кортежу. Зріз для кортежу отримують так само, як і зріз для списку.
- Визначити кількість елементів у кортежі можна за допомогою функції `len()`. Індекс елемента в кортежі визначається методом `index()`, а кількість елементів із заданим значенням можна визначити за допомогою методу `count()`.



Розділ 5

Множини, словники і текст

Читання — це не мистецтво, це
наука.

Дж. Буш (молодший)

У цьому розділі ми продовжуємо обговорювати ті типи даних, які можна було б назвати множинними, оскільки їхня концепція близька до теорії впорядкованих і невпорядкованих множин. У попередньому розділі відбулося наше знайомство зі списками й кортежами. Тому деяке уявлення про те, яким же чином великі масиви даних можуть реалізовуватися у програмному коді в Python, читач уже має. І хоча концепція списків дозволяє просто й ефективно розв'язувати широкий спектр задач, у деяких випадках корисними можуть виявитися й інші програмні конструкції — такі, наприклад, як *множини* й *словники*. Про них ітиметься в цьому розділі. Також ми детальніше обговоримо *текст*.

Множини

Я й уявлення не мав, що у нас
так багато зброї. Навіщо вона нам?

Дж. Буш (молодший)

Множина — це невпорядкований набір унікальних елементів. У цьому разі **невпорядкований** означає, що порядок розміщення елементів у множині значення не має. Щодо *унікальності*, то у множині не може бути два однакових елементи (однакові елементи розглядаються як один елемент). Фактично, якщо у нас є деякий елемент, то щодо тієї або іншої множини ми можемо лише сказати, входить елемент у множину чи ні. Говорити про позиції елемента всередині множини немає сенсу.



Умова унікальності елементів у множині має наслідки: елементами множини можуть бути значення так званих **незмінних типів** — наприклад, числа або текст. Зрозуміло, що при цьому елементи однієї множини можуть належати до різних типів даних (у межах припустимих типів).

Для створення множини використовують функцію `set()`, аргументом якій передається список зі значеннями, які включаються у множину. Також можна сформувати множину за допомогою пари фігурних дужок, усередині яких через кому перераховуються елементи множини. Однак варіант із функцією `set()` все ж кращий, оскільки за допомогою фігурних дужок задають (правда, у дещо іншому форматі) не тільки множини, але й *словники* (їх буде описано трохи пізніше).



До вищесказаного можна додати, що порожні фігурні дужки інтерпретують як порожній список, а не як порожню множину. Щоб створити порожню множину, використовуємо інструкцію `set()`.

Невеликий приклад створення множин наведено в лістингу 5.1.

Лістинг 5.1. Створення множин

```
# Вихідний список
A=[1, 30, "text", True, 30, 100, False]
# Відображаємо вміст списку
print("Список А:", A)
# На основі списку створюємо множину
B=set(A)
# Відображаємо вміст множини
print("Множина В:", B)
# Створюємо ще одну множину
C={1, 30, "text", True, 30, 100, False}
# Відображаємо вміст множини
print("Множина С:", C)
# Перевіряємо рівність множин
print("Рівність множин:", B==C)
# Входження елемента 1 у множину С
print("Елемент 1 у множині С:", 1 in C)
```

У результаті виконання програмного коду отримуємо таке:

Результат виконання програми (з лістингу 5.1)

```
Список А: [1, 30, 'text', True, 30, 100, False]
Множина В: {False, 1, 100, 'text', 30}
Множина С: {False, True, 100, 'text', 30}
Рівність множин: True
Елемент 1 у множині С: True
```

У цьому прикладі спочатку за допомогою команди `A=[1, 30, "text", True, 30, 100, False]` ми створюємо список А з декількома елементами різного типу. Для зручності список відображаємо у вікні виводу

(команда `print("Список А:", A)`). Потім на основі цього списку командою `B=set(A)` створюємо множину (використовуємо функцію `set()`, аргументом їй передаємо список `A`). Вміст множини `B` — а це елементи множини, — відображаємо у вікні виводу за допомогою команди `print("Множина В:", B)`. Що ми бачимо? По-перше, у списку `A` були числові елементи зі значенням 30, що збігаються. У множині `B` елемент із таким значенням залишився всього один. По-друге, у множині `B`, по-рівняно зі списком `A`, кудись зник елемент зі значенням `True`. Пояснення практично таке ж, як і в попередньому випадку. Річ у тім, що в Python логічні значення є підмножиною множини цілих чисел, тому `True` інтерпретується як 1. Елемент 1 у списку `A` теж є. Тому з двох елементів `True` і 1 «виживає» тільки один із них — у цьому разі це елемент 1. По-третє, у множині `B`, по-рівняно зі списком `A`, порядок елементів змінився. Але ця обставина взагалі не повинна хвилювати, оскільки для множини порядок слідування елементів значення не має. Точніше, такого поняття, як місце елемента у множині, взагалі не існує. Має значення тільки те, входить елемент у множину чи ні.

За допомогою команди `C={1,30,"text",True,30,100,False}` створюємо ще одну множину. За ідеєю, набір елементів, які перераховано через кому у фігурних дужках такий же, як і набір елементів у списку `A`, на основі якого раніше створювалася множина `B`. Логічно очікувати, що множини `B` і `C` будуть збігатися (оскільки створювалися на основі однакових наборів елементів). Але якщо за допомогою команди `print("Множина С:", C)` відобразити вміст множини `C` і порівняти з умістом множини `B`, можуть виникнути сумніви щодо справедливості твердження про рівність множин.



Дві множини вважають рівними одна одній, якщо вони складаються з однакових елементів. Зокрема, якщо відомо, що дві множини рівні, то це означає, що кожен елемент із першої множини є у другій множині, а кожен елемент із другої множини також представлено й у першій множині.

Конкретніше: у множині `B` є елемент 1, але немає елемента `True`, а в множині `C` є елемент `True`, але немає елемента 1. Однак, якщо згадати, що логічне значення `True` насправді еквівалентне цілочисловому значенню 1, то проблема, у принципі, знімається. Усе ж ми перевіряємо рівність множин `B` і `C`, для чого використовуємо команду

`print("Рівність множин:", B==C)`. Для порівняння множин на предмет рівності ми використали оператор `==`. Результатом інструкції `B==C` є значення `True`, що буває тільки тоді, коли порівнювані за допомогою оператора `==` множини містять однакові набори елементів. Ще ми перевіряємо окремо наявність елемента 1 у множині C (нагадаємо, що формально там числового елемента 1 немає, але є логічне значення `True`). Для перевірки входження елемента в множину використовують оператор `in`. Результатом команди вигляду `element in множина` буде `True`, якщо елемент входить у множину, і `False`, якщо такого елемента у множині немає. У команді `print("Елемент 1 у множині C:", 1 in C)` нами використано інструкцію `1 in C`, результатом якої є `True`. Нескладно здогадатися, що під час перевірки наявності 1 у множині C як одиничний числовий елемент інтерпретується логічне значення `True`.

Серед методів і функцій, що використовуються в роботі з множинами, чимало нам уже відомі. Так, кількість елементів у множині визначають за допомогою функції `len()` (множина передається аргументом функції). Для створення копії множини використовують метод `copy()` (метод без аргументів викликається з копійованої множини).



Множина може містити тільки елементи **незмінних** типів. Тому, наприклад, список елементом множини бути не може. Не може бути елементом множини інша множина. Як наслідок, проблема створення поверхневих або повних копій, із якою ми зустрілися під час обговорення списків, для множин є неактуальною.

Разом із тим, якщо спробувати створити копію множини за допомогою оператора присвоювання — тобто, просто присвоївши одній змінній значення іншої змінної (яка посилається на множину), то отримаємо дві змінні, які посилаються на одну й ту ж множину.

Перевірити, чи посилаються змінні на один і той самий об'єкт (у тому числі, таким об'єктом може бути множина) можна за допомогою оператора `is`.

Щоб додати елемент у множину, з неї викликають метод `add()`, аргументом якому передають елемент, що додають. Зворотну процедуру, тобто видалення елемента з множини, виконують за допомогою методів `remove()` або `discard()`. Методи викликаються з множини, а як аргумент передається елемент, який потрібно видалити з множини. Головна

відмінність між методами в тому, як вони «реагують» на ситуацію, коли елемента, який видаляють, у множині немає: під час використання методу `remove()` виникає помилка, а метод `discard()` помилку не викликає — у цьому разі просто нічого не відбувається.



Щоб повністю очистити множину (видалити всі елементи з множини), можна скористатися методом `clear()`.

Список операцій, в основному математичного характеру, які можуть виконуватися з множинами і для яких передбачено спеціальні методи і/або оператори.



Тут незайвим буде освіжити в пам'яті деякі поняття з теорії множин. Отже, припустімо, що є дві множини, які позначимо як A і B . **Об'єднанням** двох множин A і B називають таку множину $A \cup B$, яка містить у собі всі елементи множини A і всі елементи множини B . Наприклад, якщо $A = \{1, 2, 3, 4\}$ і $B = \{3, 4, 5, 6\}$, то $A \cup B = \{1, 2, 3, 4, 5, 6\}$. Результат отримуємо, об'єднавши в одну множину елементи з множин A і B . При цьому «спільні» елементи (які є й у множині A , і в множині B — а в нашому випадку, це числа 3 і 4) включаються тільки один раз (тобто не дублюються). Причина в тому, що, за визначенням, множина складається з унікальних елементів. Два однакових елементи у множину входити не можуть.

Перетином множин A і B називають таку множину $A \cap B$, яка складається з елементів, що входять одночасно як у множину A , так і в множину B . Наприклад, якщо $A = \{1, 2, 3, 4\}$ і $B = \{3, 4, 5, 6\}$, то $A \cap B = \{3, 4\}$. Оскільки, за визначенням, перетин множин складається зі спільних елементів цих множин, а в нашому випадку спільними елементами для множин A і B є числа 3 і 4, то саме ці елементи формують множину-результат.

Різницею множин A і B називають множину $A \setminus B$, яка складається з елементів множини A , які при цьому не є елементами множини B . Наприклад, якщо $A = \{1, 2, 3, 4\}$ і $B = \{3, 4, 5, 6\}$, то $A \setminus B = \{1, 2\}$. Тут результат формується так: беремо елементи з множини A , за винятком тих, що входять також і в множину B (це числа 3 і 4). Тому в підсумку отримуємо множину, яка складається з чисел 1 і 2.

Симетричною різницею множин A і B називають таку множину $A \Delta B$, яка складається з елементів множини A , що не входять у множину B , і з еле-

ментів множини B , що не входять у множину A . Іншими словами, за визначенням, симетрична різниця множин A і B – це об'єднання множин $A \cup B$, за винятком їхніх спільних елементів (перетин множин $A \cap B$), тобто має місце співвідношення $A \Delta B = (A \cup B) \setminus (A \cap B)$. Наприклад, якщо $A = \{1, 2, 3, 4\}$ і $B = \{3, 4, 5, 6\}$, то $A \Delta B = \{1, 2, 5, 6\}$. Робимо так: беремо всі елементи з множин A і B , за винятком тих із них, що одночасно входять як у множину A , так і в множину B (числа від 1 до 6 включно). Спільні елементи для двох множин – числа 3 і 4. Тому результат – множина чисел 1, 2, 5 і 6.

Для обчислення об'єднання, перетину й різниці (у тому числі й симетричної) множин у Python передбачено цілу групу спеціальних методів (і операторів). Розглянемо їх. Далі через A і B позначено деякі множини.

Для обчислення *об'єднання* множин можна використовувати оператор `|` (оператор об'єднання множин — якщо операнди є множинами) або метод `union()`, який викликається з першої множини, а аргументом йому передається друга множина. Відповідні команди мають вигляд $A | B$ або $A.union(B)$ (можна також скористатися командою $B.union(A)$ — результат буде таким самим). Самі множини A і B при цьому не змінюються, а результатом є нова множина, що складається з елементів множин A і B . Якщо нам не потрібно в результаті об'єднання множин створювати нову множину, а ми хочемо замість цього «розширити», наприклад, множину A за рахунок елементів множини B , корисним буде метод `update()`. У результаті виконання команди $A.update(B)$ створюється об'єднання множин A і B , а результат записується в змінну A . Такого ж ефекту можемо досягти, скажімо, за допомогою команд $A=A|B$ або $A|=B$ (скорочена форма для оператора об'єднання множин `|`). Приклади виконання операції об'єднання множин наведено в лістингу 5.2.

Лістинг 5.2. Об'єднання множин

```
# Перша множина (множина A)
A={1,2,3,4}
# Відображаємо вміст множини
print("Множина A:",A)
# Друга множина (множина B)
B={3,4,5,6}
# Відображаємо вміст множини
print("Множина B:",B)
```

```

# Об'єднання множин (множина C)
C=A|B
# Відображаємо результат
print("Множина C=A|B:",C)
# Об'єднання множин
print("Множина A.union(B):",A.union(B))
print("Множина B.union(A):",B.union(A))
# Змінюємо множину A
A.update(B)
# Перевіряємо результат
print("Множина A:",A)
# Змінюємо множину B
B=B|{-1,-2,-3}
# Перевіряємо результат
print("Множина B:",B)
# Змінюємо множину C
C|={7,8,9}
# Перевіряємо результат
print("Множина C:",C)

```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 5.2)

```

Множина A: {1, 2, 3, 4}
Множина B: {3, 4, 5, 6}
Множина C=A|B: {1, 2, 3, 4, 5, 6}
Множина A.union(B): {1, 2, 3, 4, 5, 6}
Множина B.union(A): {1, 2, 3, 4, 5, 6}
Множина A: {1, 2, 3, 4, 5, 6}
Множина B: {3, 4, 5, 6, -2, -3, -1}
Множина C: {1, 2, 3, 4, 5, 6, 7, 8, 9}

```

Для обчислення *перетину* двох множин можна використовувати метод `intersection()` або оператор `&` (оператор перетину множин — якщо опера́нди є множинами): формат команд виглядає як `A.intersection(B)` або `A&B`. Як і в разі об'єднання множин, самі множини A і B не змінюються, а порядок, в якому вказані множини, значення не має (команди

`A.intersection(B), B.intersection(A), A&B і B&A` з точки зору кінцевого результату еквівалентні).

Якщо ми хочемо, щоб під час обчислення перетину множин замість створення нової множини результат записувався в одну з вихідних множин, використовуємо метод `intersection_update()`. Наприклад, у результаті виконання команди `A.intersection_update(B)` обчислюється перетин множин `A` і `B` і посилання на отриману множину записується в змінну `A`. Такого ж результату досягаємо за допомогою команд `A=A&B` або `A&=B` (скорочена форма для оператора перетину множин `&`). У лістингу 5.3 наведено приклади обчислення перетину множин.

Лістинг 5.3. Перетин множин

```
# Перша множина (множина А)
A={1, 2, 3, 4}
# Відображаємо вміст множини
print("Множина А:",A)
# Друга множина (множина В)
B={3, 4, 5, 6}
# Відображаємо вміст множини
print("Множина В:",B)
# Перетин множин (множина С)
C=A&B
# Відображаємо результат
print("Множина С=A&B:",C)
# Перетин множин
print("Множина A.intersection(B):",A.intersection(B))
print("Множина B.intersection(A):",B.intersection(A))
# Змінюємо множину А
A.intersection_update(B)
# Перевіряємо результат
print("Множина А:",A)
# Змінюємо множину В
B=B&{4, 6, 8, 10}
# Перевіряємо результат
print("Множина В:",B)
```

```
# Змінюємо множину С
C&={1,2,3}
# Перевіряємо результат
print("Множина C:",C)
```

У результаті виконання наведеного програмного коду отримуємо таке:

Результат виконання програми (з лістингу 5.3)

```
Множина A: {1, 2, 3, 4}
Множина B: {3, 4, 5, 6}
Множина C=A&B: {3, 4}
Множина A.intersection(B): {3, 4}
Множина B.intersection(A): {3, 4}
Множина A: {3, 4}
Множина B: {4, 6}
Множина C: {3}
```

Різницю множин обчислюють методом `difference()` або за допомогою оператора `-` (оператор «мінус» для operandів-множин інтерпретується як оператор обчислення різниці множин). У результаті обчислення виразів вигляду `A.difference(B)` і `A-B` повертається нова множина, яке складається з тих елементів множини `A`, які не є елементами множини `B` (очевидно, що тут має значення порядок, в якому вказані множини). Для того щоб результат обчислення різниці множин записувався в змінну `A`, використовуємо команди `A.difference_update(B)` (тут із множини `A` викликається метод `difference_update()` із аргументом — множиною `B`), `A=A-B` або `A-=B` (скорочена форма для оператора обчислення різниці множин). Приклади обчислення різниці множин наведено в лістингу 5.4.

Лістинг 5.4. Різниця множин

```
# Перша множина (множина А)
A={1,2,3,4}
# Відображаємо вміст множини
print("Множина A:",A)
# Друга множина (множина В)
B={3,4,5,6}
```

```
Phyton .....  
  
# Відображаємо вміст множини  
print("Множина B:",B)  
# Різниця множин (множина C)  
C=A-B  
# Відображаємо результат  
print("Множина C=A-B:",C)  
# Різниця множин  
print("Множина A.difference(B) :",A.difference(B))  
print("Множина B.difference(A) :",B.difference(A))  
# Змінюємо множину A  
A.difference_update(B)  
# Перевіряємо результат  
print("Множина A:",A)  
# Змінюємо множину B  
B=B-{4,6,8,10}  
# Перевіряємо результат  
print("Множина B:",B)  
# Змінюємо множину C  
C={1,3,5}  
# Перевіряємо результат  
print("Множина C:",C)
```

Результат виконання програмного коду буде таким:

Результат виконання програми (з листингу 5.4)

```
Множина A: {1, 2, 3, 4}  
Множина B: {3, 4, 5, 6}  
Множина C=A-B: {1, 2}  
Множина A.difference(B): {1, 2}  
Множина B.difference(A): {5, 6}  
Множина A: {1, 2}  
Множина B: {3, 5}  
Множина C: {2}
```

Щоб обчислити *симетричну різницю* множин, використовуємо метод `symmetric_difference()`. Викликається метод із першої множини, друга множина передається як аргумент, а результатом є нова множина.

Множина, отримана як значення виразу `A.symmetric_difference(B)`, складається з елементів множин `A` і `B`, але тільки тих, які належать лише одній із множин. Такий же результат отримуємо за допомогою інструкції `A^B`. Тут ми використовуємо оператор `^` (якщо операндами є множини, то оператор обчислює симетричну різницю). Якщо скористатися командою `A=A^B` або `A^=B` (скорочена форма для обчислення симетричної різниці), то буде обчислено симетричну різницю множин `A` і `B`, а результат записано в змінну `A`. Цій же меті служить і метод `symmetric_difference_update()`. Приклади обчислення симетричної різниці наведено в лістингу 5.5.

Лістинг 5.5. Симетрична різниця множин

```
# Перша множина (множина A)
A={1,2,3,4}
# Відображаємо вміст множини
print("Множина A:",A)
# Друга множина (множина B)
B={3,4,5,6}
# Відображаємо вміст множини
print("Множина B:",B)
# Симетрична різниця множин (множина C)
C=A^B
# Відображаємо результат
print("Множина C=A^B:",C)
# Різниця множин
print("Множина A.symmetric_difference(B):",A.symmetric_
difference(B))
print("Множина B.symmetric_difference(A):",B.symmetric_
difference(A))
# Змінюємо множину A
A.symmetric_difference_update(B)
# Перевіряємо результат
print("Множина A:",A)
# Змінюємо множину B
B=B^{4,6,8,10}
# Перевіряємо результат
print("Множина B:",B)
```

Phyton

```
# Змінюємо множину С  
C^={1, 3, 5}  
# Перевіряємо результат  
print("Множина C:", C)
```

Після виконання цього програмного коду отримуємо такий результат:

Результат виконання програми (з лістингу 5.5)

```
Множина A: {1, 2, 3, 4}  
Множина B: {3, 4, 5, 6}  
Множина C=A^B: {1, 2, 5, 6}  
Множина A.symmetric_difference(B): {1, 2, 5, 6}  
Множина B.symmetric_difference(A): {1, 2, 5, 6}  
Множина A: {1, 2, 5, 6}  
Множина B: {3, 5, 8, 10}  
Множина C: {2, 3, 6}
```



Оператори `|`, `&` і `^`, які згадувалися вище при обчисленні об'єднання, перетину і симетричної різниці множин, нам уже знайомі (у всякому разі, візуально). До цього вони були представлені світу як побітові оператори. Фактично багато чого залежить від операндів. Наприклад, оператори `|`, `&` і `^` допускають використовувати для виконання логічних операцій: оператор `|` обчислює логічне **або** (аналог оператора `or`), оператор `&` обчислює логічне *i* (аналог оператора `and`), а оператор `^` обчислює логічне **виключне або** (аналог оператора `xor`). Усе це відбувається, якщо операндами для операторів `|`, `&` і `^` вказати логічні значення `True` або `False`. Виправдання такому «демократизмові» очевидне. Логічні значення є підмножиною цілих чисел. Конкретніше, логічне значення `True` інтерпретується як 1, а логічне значення `False` інтерпретується як 0. Для цих цілих чисел виконуються побітові операції, результатом яких можуть бути знову ж таки числа 1 або 0, які інтерпретуються як логічні значення `True` і `False` відповідно.

Якщо «дивитись у корінь», то операції з множинами, логічні операції та побітові операції насправді однотипні. Аналогію проілюструємо на прикладі логічних операцій. У цьому разі під `True` слід розуміти належність елемента множині, а під `False` – відсутність елемента в множині. Наприклад, вираз вигляду `A | B` з логічними операндами `A` і `B` відповідає логічному **або**: результатом буде `True`, якщо хоча б один із операндів дорівнює `True`. Для

множин A і B вираз $A \mid B$ – це об'єднання множин. Результатом операції $A \mid B$ є множина, що складається з деяких елементів. Візьмімо тепер довільний елемент. Він може належати множині $A \mid B$ (аналог значення `True` для результату логічного виразу), а може не належати множині $A \mid B$ (аналог значення `False` для результату логічного виразу). Елемент належить множині $A \mid B$, якщо він належить хоча б одній із множин A (аналог значення `True` для операнда A в логічному виразі) або B (аналог значення `True` для операнда B в логічному виразі).

Для логічного *i* значення виразу $A \& B$ із логічними operandами A і B дорівнює `True`, тільки якщо обидва операнди A і B дорівнюють `True`. Для множин A і B результатом виразу $A \& B$ (перетин множин) є множина, що складається з елементів, що належать одночасно A і B (аналог значень `True` для operandів у логічному виразі).

Нарешті, логічне **виключне або** для логічних operandів A і B (вираз $A \wedge B$) дає значення `True`, якщо один і тільки один із operandів дорівнює `True`. Симетрична різниця $A \wedge B$ для множин A і B дає в результаті множину, що складається з елементів, які належать одній і тільки одній із множин A і B .

Окрім перерахованих операцій із множинами, часто доводиться мати справу з операціями відношення (що виконуються в контексті роботи з множинами). Як порівнювати множини на предмет рівності і як перевіряти, чи входить елемент до складу множини, ми вже знаємо. У першому випадку корисним буде оператор `==`, а у другому — оператор `in`.



Дві множини A і B вважають **рівними** (позначається як $A = B$), якщо кожний елемент множини A входить у множину B , а кожний елемент множини B входить у множину A .

Множина A є **підмножиною** множини B (позначається як $A \subset B$ або $A \subseteq B$, якщо множини A і B можуть збігатися), якщо кожний елемент множини A є елементом множини B .

Як оператори відношення використовуються стандартні оператори порівняння, а для деяких є ще додаткові методи. Основні оператори і відповідні їм операції описано в таблиці 5.1. Через A і B позначено деякі множини, над якими виконують операції.

Таблиця 5.1. Основні операції відношення для множин

Оператор і/або метод	Опис
<	Результатом виразу $A < B$ є True, якщо всі елементи множини A входять у множину B (множина A є підмножиною множини B), причому обидві множини не рівні. А якщо інакше, то значення виразу дорівнює False
\leq або <code>issubset()</code>	Результатом виразу $A \leq B$ (або виразу <code>A.issubset(B)</code>) є True, якщо всі елементи множини A входять у множину B (множина A є підмножиною множини B), причому допускається рівність множин. А якщо інакше, то значення виразу дорівнює False
>	Результатом виразу $A > B$ є True, якщо всі елементи множини B входять у множину A (множина B є підмножиною множини A), причому обидві множини не рівні. А якщо інакше, то значення виразу дорівнює False
\geq або <code>issuperset()</code>	Результатом виразу $A \geq B$ (або виразу <code>A.issuperset(B)</code>) є True, якщо всі елементи множини B входять у множину A (множина B є підмножиною множини A), причому допускається рівність множин. А якщо інакше, то значення виразу дорівнює False
$==$	Результатом виразу $A == B$ є True, якщо всі елементи множини A входять у множину B, а всі елементи множини B входять у множину A (рівність множин). А якщо інакше, то значення виразу дорівнює False
<code>isdisjoint()</code>	Результатом виразу <code>A.isdisjoint(B)</code> є значення True, якщо перетин множин A і B є порожньою множиною (тобто якщо у множин A і B немає спільних елементів). А якщо інакше, то значення виразу дорівнює False
<code>in</code>	Результатом виразу <code>a in A</code> є значення True, якщо елемент a входить у множину A. А якщо інакше, то значення виразу дорівнює False



Порожньою називається множина, яка не містить жодного елемента (звичайно позначається як \emptyset).

Слід зазначити, що множини можна використовувати в операторах циклу для перебiranня значень змінної циклу, як це було зі списками. Єдине, про що слід пам'ятати, — на відміну від списку, у множині порядок елементів не фіксований. Якого саме значення змінна циклу набуває на тій чи іншій ітерації, сказати вкрай проблематично.



Для створення множин можна використовувати **генератори множин**. Принцип той самий, що й у генераторах списків, тільки для множин генератор беруть не в прямокутні, а у фігурні дужки.

Як ілюстрацію до використання множин розглянемо розв'язання програмними методами такої задачі: необхідно визначити, які числа в діапазоні значень від 1 до 100 задовольняють такі умови:

- числа в результаті ділення на 5 дають в остачі 2 або 4;
- числа в результаті ділення на 7 дають в остачі 3;
- в результаті ділення чисел на 3 в остачі отримуємо число, відмінне від 1.

Це — нескладна задача, і розв'язувати її можна різними способами. Найпростіша й очевидна ідея, яка спадає на думку, — послідовно перебрати всі натуральні числа від 1 до 100 і для кожного числа перевірити виконання перерахованих вище умов. Але нас, зрозуміло, цікавить не стільки кінцевий результат, скільки методологічний бік питання. Тому для розв'язання цієї задачі в Python ми будемо «відштовхуватися» від теорії множин і можливостей мови Python.



Щоб зрозуміти алгоритм розв'язання задачі, необхідно взяти до уваги такі обставини.

Множину натуральних чисел від 1 до 100 позначимо як E . Множину чисел, які при діленні на 5 дають в остачі 2, позначимо як A_1 . Множину чисел, які при діленні на 5 дають в остачі 4, позначимо як A_2 . Числа, які при діленні на 7 дають в остачі 3, позначимо як множину B . Множину чисел, які при діленні на 3 дають в остачі 1, позначимо як множину C .

ленні на 3 дають в остачі 1, позначимо як C . У всіх перерахованих випадках ідеться про числа, які не перевищують 100. Тому всі ці множини є підмножинами множини E .

Множина чисел, які при діленні на 5 дають в остачі 2 або 4, позначимо як A . Очевидно, що $A = A_1 \cup A_2$. Множина чисел, які при діленні на 5 дають в остачі 2 або 4, а при діленні на 7 дають в остачі 3, визначається виразом $A \cap B$, оскільки такі числа повинні одночасно належати як множині A (остача 2 або 4 при діленні на 5), так і множині B (остача 3 при діленні на 7). Якщо ми накладаємо умову, щоб такі числа ще й при діленні на 3 не давали в остачі 1, то це означає, що числа не повинні належати множині C (остача 1 при діленні на 3). Тому з множини $A \cap B$ слід «відняти» множину C . Таким чином, множина $D = (A \cap B) \setminus C$ містить потрібні числа — тобто числа, які задовільняють усі перераховані умови.

Відповідний програмний код наведено в лістингу 5.6.

Лістинг 5.6. Використання множин

```
# Верхня межа
n=100
# Множина натуральних чисел
E={s for s in range(1,n+1)}
# Множина чисел, які при діленні
# на 5 дають в остачі 2
A1={s for s in E if s%5==2}
# Множина чисел, які при діленні
# на 5 дають в остачі 4
A2={s for s in E if s%5==4}
# Множина чисел, які при діленні
# на 5 дають в остачі 2 або 4
A=A1|A2
# Множина чисел, які при діленні
# на 7 дають в остачі 3
B={s for s in E if s%7==3}
# Множина чисел, які при діленні
# на 3 дають в остачі 1
C={s for s in E if s%3==1}
```

```
# Множина чисел, які при діленні на 5 дають в остачі 2
# або 4, при діленні на 7 дають в остачі 3, а при діленні
# на 3 не дають в остачі 1
D=(A&B)-C
# Відображаємо результат
print("Наведені нижче числа від 1 до",n)
print("1) при діленні на 5 дають в остачі 2 або 4;")
print("2) при діленні на 7 дають в остачі 3;")
print("3) при діленні на 3 не дають в остачі 1:")
print(D)
```

У цьому разі отримуємо такий результат:

Результат виконання програми (з лістингу 5.6)

Наведені нижче числа від 1 до 100
 1) при діленні на 5 дають в остачі 2 або 4;
 2) при діленні на 7 дають в остачі 3;
 3) при діленні на 3 не дають в остачі 1:
 {24, 17, 59, 87}

У змінну n записується значення для верхньої межі числового діапазону. Множина натуральних чисел зі значеннями в діапазоні від 1 до n формується командою E={s for s in range(1,n+1)}. Тут ми використали генератор множини: у фігурних дужках інструкція s for s in range(1,n+1) означає, що елемент множини визначається змінною s, яка «пробігає» значення натурального ряду від 1 до n.

Множину A1 створюємо за допомогою команди A1={s for s in E if s%5==2}. Тепер у генераторі множини змінна s перебирає значення елементів із множини E. Елемент заноситься до множини A1 за умови s%5==2, тобто якщо остача від ділення s на 5 дорівнює 2. Аналогічно команди A2={s for s in E if s%5==4}, B={s for s in E if s%7==3} і C={s for s in E if s%3==1} створюють інші множини. Множину чисел, які при діленні на 5 дають в остачі 2 або 4, обчислюємо шляхом об'єднання множин A1 і A2 — за допомогою команди A=A1|A2. Зрештою, за допомогою команди D=(A&B)-C знаходимо потрібний результат.

Комусь, можливо, більш удалим видається інший підхід, у якому множина-результат формується одразу за допомогою генератора множин. Цей спосіб розв'язання задачі наведено в лістингу 5.7.

Лістинг 5.7. Генерування множини

```
# Верхня межа
n=100
# Множина чисел, які при діленні
# на 5 дають в остачі 2 або 4, при діленні на 7
# дають в остачі 3, а при діленні на 3 не дають
# в остачі 1
D={s for s in range(1,n+1) if (s%5==2 or s%5==4) and
s%7==3 and s%3!=1}
# Відображаємо результат
print("Наведені нижче числа від 1 до",n)
print("1) при діленні на 5 дають в остачі 2 або 4;")
print("2) при діленні на 7 дають в остачі 3;")
print("3) при діленні на 3 не дають в остачі 1:")
print(D)
```

Результат виконання цього програмного коду такий же, як і в попередньому випадку:

Результат виконання програми (з лістингу 5.7)

```
Наведені нижче числа від 1 до 100
1) при діленні на 5 дають в остачі 2 або 4;
2) при діленні на 7 дають в остачі 3;
3) при діленні на 3 не дають в остачі 1:
{24, 17, 59, 87}
```

Тут ми результат сформували фактично однією командою `D={s for s in range(1,n+1) if (s%5==2 or s%5==4) and s%7==3 and s%3!=1}`, у якій у генераторі множин змінна `s` «пробігає» значення натурального ряду від 1 до n. Але відповідне число включається у створювану множину тільки тоді, якщо виконано складну умову `(s%5==2 or s%5==4) and s%7==3 and s%3!=1`. У цій умові за допомогою логічних операторів `or` (*логічне або*) і `and` (*логічне і*) об'єднані декілька логічних виразів:

одночасно повинні виконуватися умови ($s \% 5 == 2$ or $s \% 5 == 4$), $s \% 7 == 3$ і $s \% 3 != 1$, причому умова $s \% 5 == 2$ or $s \% 5 == 4$ полягає в тому, що виконується хоча б одна з умов $s \% 5 == 2$ або $s \% 5 == 4$.

З іншого боку, можна скористатися й більш «класичним» підходом. Приклад показано в лістингу 5.8.

Лістинг 5.8. Створення множини через цикл

```
# Верхня межа
n=100
# Порожня множина
D=set()
# Формується множина чисел, які при діленні
# на 5 дають в остачі 2 або 4, при діленні на 7
# дають в остачі 3, а при діленні на 3 не дають
# в остачі 1
for s in range(1,n+1):
    if s%5==2 or s%5==4:
        if s%7==3:
            if s%3!=1:
                D.add(s)
# Відображаємо результат
print("Наведені нижче числа від 1 до",n)
print("1) при діленні на 5 дають в остачі 2 або 4;")
print("2) при діленні на 7 дають в остачі 3;")
print("3) при діленні на 3 не дають в остачі 1:")
print(D)
```

Результат отримаємо такий:

Результат виконання програми (з лістингу 5.8)

Наведені нижче числа від 1 до 100

- 1) при діленні на 5 дають в остачі 2 або 4;
 - 2) при діленні на 7 дають в остачі 3;
 - 3) при діленні на 3 не дають в остачі 1:
- {24, 17, 59, 87}

За такого підходу ми спочатку створюємо командою `D=set()` порожню множину, а потім за допомогою оператора циклу і групи вкладених умовних операторів додаємо до списку нові елементи — зрозуміло, якщо ці елементи задовольняють усі критерії (умови в умовних операторах). Для додавання елемента в множину задіяний метод `add()`.



Окрім типу `set` (**множина**), у мові Python є тип `frozenset` (**незмінювана множина**). Головна відмінність незмінюваної множини від звичайної множини полягає в тому, що незмінювану множину, як нескладно згадатися, не можна змінити. Створюють незмінювану множину за допомогою функції `frozenset()`, аргументом якій передають, наприклад, список або текстовий рядок. Чимало методів, застосовних до звичайних множин, можуть застосовуватися до незмінних множин — безумовно, йдеться про методи, що не змінюють структуру вихідної множини.

Далі ми розглянемо ще один корисний тип даних — а саме, словники.

Словники

Хочу подякувати вам за значення, яке ви виявили до освіти і грамотності.

Дж. Буш (молодший)

У певному сенсі *словники* можна розглядати як дещо середнє між списками і множинами. Словник є невпорядкованим набором елементів. Однак, на відміну від множини, кожен елемент словника «ідентифікований» спеціальним чином. А саме, з кожним елементом словника зіставляється *ключ*, який однозначно ідентифікує елемент у словнику. Ситуація трохи схожа на індексацію елементів у списку. Але в списку індекси — цілочислові ранжирувані значення. Ключі ж в словнику не обов'язково повинні бути числами: окрім чисел, це можуть бути, наприклад, текстові рядки, списки або кортежі. Тобто з певною натяжкою можна думати про словники як про списки, але з нечисловими (у загальному випадку) індексами.

Для створення словника використовують функцію `dict()`. Як аргумент функції можуть передаватися, розділені комами, вирази вигляду `ключ=значення`. Також можна передати список, елементами якого є списки (або кортежі) по два елементи в кожному: ключ і відповідне йому значення.



Якщо функції `dict()` під час виклику не передати аргументи, буде створено порожній словник. Якщо аргументом функції `dict()` передають уже існуючий словник, то в результаті буде створено копію такого словника (правда, поверхневу — тобто для таких укладених елементів словника, як списки і словники, виконується копіювання посилань, але не копіювання елементів).

Можна створити словник, указавши у фігурних дужках через коми вирази вигляду **ключ:значення** (ключ і відповідне йому значення розділяють двокрапкою).

Звертання до елемента словника виконується формально так само, як і звертання до елемента списку: за допомогою квадратних дужок після імені словника, але тільки у випадку словника у квадратних дужках указують не індекс, а ключ. Іншими словами, якщо ми маємо справу зі словником, то дізнається, яке значення елемента із зазначеним ключем, можемо за допомогою інструкції вигляду **словник[ключ]**. Причому таку інструкцію використовують не тільки для того, щоб прочитати значення елемента, але й щоб змінити це значення. Невеликі приклади створення словників наведено в лістингу 5.9.

Лістинг 5.9. Створення словників

```
# Список для формування словника
A=[["Шевченко Т.Г.", "Кобзар"], ["Франко І.Я.", "Лис Микита"],
    ["Сковорода Г.С.", "Сад божественних пісень"] ]
# Створюємо словник на основі списку
writers=dict(A)
# Відображаємо вміст словника
print("Словник:")
print(writers)
# Звертання до елемента словника за ключем
print("Франко написав казку:",writers["Франко І.Я."])
# Змінюємо значення елемента словника
writers["Франко І.Я."]="Украдене щастя"
# Перевіряємо вміст словника
print("Словник після зміни елемента:")
print(writers)
# Додаємо до словника новий елемент
writers["Костенко Л.В."]="Записки українського
самашедшого"
# Перевіряємо вміст словника
print("Словник після додавання елемента:")
print(writers)
print()
```

```

# Перебір елементів словника за ключем
print("Автори та їхні твори.")
for s in writers.keys():
    print("Автор:", s)
    print("Твіп:", writers[s])
    print()
# Створюємо новий словник
lights=dict(чорне="рух заборонено", жовте="увага",
зелене="рух дозволено")
# Перевіряємо вміст словника
print("Новий словник:")
print(lights)
# Значення ключа
color="зелене"
# Звертання до елемента словника за ключем
print("Якщо горить", color, "світло, то", lights[color] + "!")
print()
# Створюємо ще один словник
girls={(90, 60, 90):"Світлана", (85, 65, 89):"Юля", (92, 58, 91):"Ніна"}
# Перевіряємо вміст словника
print("Ще один словник:")
print(girls)
# Значення ключа
params=(90, 60, 90)
# Звертання до елемента словника за ключем
print(girls[params] + ":", params)

```

Нижче показано результат виконання програмного коду:

Результат виконання програми (з лістингу 5.9)

Словник:

```
{'Франко І.Я.': 'Лис Микита', 'Сковорода Г.С.': 'Сад божественних пісень', 'Шевченко Т.Г.': 'Кобзар'}
```

Франко написав казку: Лис Микита

Словник після зміни елемента:

```
{'Франко І.Я.': 'Украдене щастя', 'Сковорода Г.С.': 'Сад божественних пісень', 'Шевченко Т.Г.': 'Кобзар'}
```

Phyton
.....

Словник після додавання елемента:

{ 'Франко І.Я.': 'Украдене щастя', 'Сковорода Г.С.':
'Сад божественних пісень', 'Костенко Л.В.': 'Записки
українського самашедшого', 'Шевченко Т.Г.': 'Кобзар' }

Автори та їхні твори.

Автор: Франко І.Я.

Твір: Украдене щастя

Автор: Сковорода Г.С.

Твір: Сад божественних пісень

Автор: Костенко Л.В.

Твір: Записки українського самашедшого

Автор: Шевченко Т.Г.

Твір: Кобзар

Новий словник:

{ 'зелене': 'рух дозволено', 'жовте': 'увага', 'червоне':
'рух заборонено' }

Якщо горить зелене світло, то рух дозволено!

Ще один словник:

{ (92, 58, 91): 'Ніна', (85, 65, 89): 'Юля', (90, 60, 90):

'Світлана' }

Світлана: (90, 60, 90)

.....

У цьому програмному коді проілюстровано декілька способів створення словників, а також деякі нескладні прийоми роботи зі словниками. Спочатку за допомогою команди A=[["Шевченко Т.Г.", "Кобзар"], ["Франко І.Я.", "Лис Микита"], ["Сковорода Г.С.", "Сад божественних пісень"]] створено список A. Він складається з трьох елементів. Кожен елемент сам є списком. У кожному внутрішньому списку — по два елементи. Перший елемент відіграє роль ключа під час створення словника, а другий елемент — безпосередньо елемент словника. У цьому разі всі значення текстові, тому й ключі словника, і безпосередньо

елементи словника будуть текстом. Для створення словника список A передаємо як аргумент функції `dict()` (команда `writers=dict(A)`). Щоб оцінити вміст словника `writers`, передаємо ім'я словника аргументом функції `print()` (команда `print(writers)`).



Як нескладно помітити із вмісту у вікні виводу, словник відображається в такому форматі: у фігурних дужках через кому подано пари значень ключа і відповідного йому елемента словника. Ключ і значення елемента розділені двокрапкою. Порядок відображення таких пар у фігурних дужках довільний. Якщо вдуматися, то стають очевидними причини такого «лібералізму»: значення елемента словника однозначно визначається за ключем і не залежить від «місця» (або позиції) елемента в словнику. Більше того, таке поняття, як позиція елемента в словнику, не має сенсу.

Як зазначалося раніше, звертання до елемента словника виконують за ключем. Наприклад, інструкція `writers["Франко І.Я."]` є звертанням до елемента словника `writers` з ключем "Франко І.Я." (елемент зі значенням "Лис Микита").

Командою `writers["Франко І.Я."]="Украдене щастя"` ми змінюємо значення елемента словника з ключем "Франко І.Я.". Якщо після цього перевірити вміст словника `writers` (що, власне, й робиться), то можна помітити зміни у вмісті словника.

Якщо в команді присвоювання значення елементу словника вказати ключ, який у словнику не подано, то до словника буде додано елемент із відповідним ключем. Наприклад, у результаті виконання команди `writers["Костенко Л.В."]="Записки українського самашедшого"` до словника `writers` додається елемент "Записки українського самашедшого" з ключем "Костенко Л.В.".

Для перебору елементів словника за ключем можна використовувати оператор циклу. Доступ до ключів словника отримують за допомогою методу `keys()`. Як результат метод повертає ітерований об'єкт (тобто який допускає процес «перебирання» вмісту) із ключами словника (того словника, з якого викликають метод). На результат виклику методу `keys()` можна дивитися як на «контейнер», в якому «містяться» ключі словника.

В операторі циклу цей контейнер указується так само, як наче це була б множина, що складається з ключів словника. Скажімо, якщо ми маємо справу з оператором циклу, в якому початкова інструкція виглядає як `for s in writers.keys()`, то це означає, що змінна `s` буде послідовно набувати значення ключів зі словника `writers` (правда, невідомо, в якій саме послідовності будуть перебиратися ключі). У тілі оператора циклу командою `print("Автор:", s)` відображається значення ключа, а командою `print("Твір:", writers[s])` відображається значення елемента з відповідним ключем. Для створення порожнього рядка у вікні виводу використовуємо команду `print()`.

Новий словник створюємо за допомогою команди `lights=dict(червоне="рух заборонено", жовте="увага", зелене="рух дозволено")`. Аргументами функції `dict()` передаємо вирази вигляду `ключ=значення`. І ключі, і значення, як і в попередньому випадку, текстові.



Зверніть увагу, що текстові значення ключів у цьому конкретному випадку вказуються без лапок.

Уміст словника перевіряємо командою `print(lights)`. Командою `color="зелене"` у змінну `color` записуємо значення ключа, після чого цю змінну використовуємо у команді `print("Якщо горить", color, "світло, то", lights[color]!"")`, щоб відобразити значення ключа і значення відповідного ключу елемента словника `lights`.

Нарешті, за допомогою команди `girls={(90, 60, 90):"Світлана", (85, 65, 89):"Юля", (92, 58, 91):"Ніна"}` створюємо ще один словник. Тут дві особливості: по-перше, елементи словника разом із ключами (розділеними двокрапкою) вказуємо у фігурних дужках. По-друге, ключами словника в даному випадку є кортежі. У цьому немає нічого дивного: як ключ словника кортеж цілком легітимний. Ілюстрацією до сказаного є команда `print(girls[params]+":", params)`, де ключем для елемента словника `girls` вказано змінну `params`, якій попередньо командою `params=(90, 60, 90)` значенням присвоєно кортеж.



Якщо при звертанні до елемента словника вказати неправильний ключ (ключ, якого в словнику немає), виникне помилка. Є метод `get()`, який дозволяє за ключем отримати значення елемента словника. При цьому, якщо вказано неіснуючий ключ (аргумент методу), помилки не буде (як метод повертається «порожнє» значення `None`). Методу `get()` можна передати другий аргумент — це значення буде повертатися як результат методу при неправильному ключі.

Окрім отримання доступу до елемента словника за ключем, існують й інші цікаві операції, які можна виконувати зі словниками. Так, ми вже знаємо, що за допомогою методу `keys()` отримують доступ до ключів словника. Доступ до значень словника отримують за допомогою методу `values()`.



Обидва методи повертають результатом об'єкти. Для методу `keys()` це об'єкт класу `dict_keys`. Метод `values()` повертає об'єкт класу `dict_values`. У цих об'єктах «заховані» відповідно ключі й значення елементів словника. Ми самі об'єкти обговорювати не плануємо. Для нас лише важливо, що ці об'єкти допускають ітерації. Тобто вміст об'єктів (ключі й значення елементів) можна послідовно «перебирати», хоча робиться це й не так «прямолінійно», як можна було б припустити. Тому про об'єкти, що повертаються методами `dict_keys` і `values()`, із певним обмеженням, можна думати як про певну віртуальну послідовність значень. Цю віртуальну послідовність можна перетворити на простіший і зрозуміліший формат: список, кортеж або множину — залежно від конкретних потреб і специфіки розв'язуваної задачі. Для цього результат методу `keys()` або `values()` передається як аргумент методу `list()` (створення списку), `tuple()` (створення кортежу) або `set()` (створення множини).

Є також метод `items()`, який повертає об'єкт класу `dict_items`, що містить кортежі з парами значень для ключів і елементів. Наведені вище зауваження стосуються й методу `items()`.

Видалити елемент зі словника можна за допомогою методу `pop()`. Як аргумент методу передається ключ елемента, який видаляють. Метод повертає значення — це значення того елемента, який видаляється зі словника. Якщо потрібно видалити елемент словника «тихо», без повернення результату, — використовують інструкцію `del`, після якої розміщують посилання на елемент, який видаляють зі словника (ім'я словника

й у квадратних дужках після імені — ключ елемента). Повністю очистити словник можна за допомогою методу `clear()`.

Як додавати елемент у словник, ми вже бачили на розглянутому вище прикладі: елементу з новим ключем просто присвоюють значення. Якщо потрібно додати одразу декілька елементів, то зручніше скористатися методом `update()`. Метод не повертає результат і викликається з того словника, в який потрібно додати нові елементи. Як аргумент методу може передаватися словник, елементи якого додаються у вихідний словник (із якого викликано метод). Також аргументом методу `update()` може бути список, через який реалізується словник, що додається, або набір розділених комами виразів вигляду `ключ=значення`. Деякі приклади використання переведених вище методів для роботи зі словниками наведено в лістингу 5.10.



Лістинг 5.10. Робота зі словниками

```
# Створюємо словник
symbols=dict([["a","перший"], ["b", "другий"]])
# Створюємо ще один словник
more_symbols=dict([["c", "третій"], ["d", "четвертий"]])
# Додаємо другий словник у перший
symbols.update(more_symbols)
# Вміст словника
print("Словник:", symbols)
# Довжина словника
print("Кількість ключів у словнику:", len(symbols))
# Доступ до елемента за ключем (ключ у словнику є)
print("Елемент із ключем 'c':", symbols.get("c", "немає такого
ключа!"))
# Перевіряємо наявність ключа в словнику
print("Наявність елемента з ключем 'c':", "c" in symbols)
# Видаляємо елемент зі словника
del symbols["c"]
# Вміст словника
print("Словник:", symbols)
# Доступ до елемента за ключем (ключа в словнику немає)
print("Елемент з ключем 'c':",
symbols.get("c", "немає такого ключа!"))
```

```

# Перевіряємо наявність ключа в словнику
print("Наявність елемента з ключем 'c':", "c" in symbols)
# Список ключів словника
print("Ключі словника:", list(symbols.keys()))
# Список значень елементів словника
print("Значення елементів словника:", list(symbols.values()))
# Вміст словника
print("Вміст словника:", list(symbols.items()))
# Видалення елемента зі словника
print("Видаляється елемент зі значенням:", symbols.pop("b"))
# Вміст словника
print("Словник:", symbols)
# Очистка словника
symbols.clear()
# Вміст словника
print("Словник:", symbols)

```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 5.10)

```

Словник: {'b': 'другий', 'c': 'третій', 'a': 'перший',
'd': 'четвертий'}
Кількість ключів у словнику: 4
Елемент із ключем 'c': третій
Наявність елемента з ключем 'c': True
Словник: {'b': 'другий', 'a': 'перший', 'd': 'четвертий'}
Елемент з ключем 'c': немає такого ключа!
Наявність елемента з ключем 'c': False
Ключі словника: ['b', 'a', 'd']
Значення елементів словника: ['другий', 'перший',
'четвертий']
Вміст словника: [(('b', 'другий'), ('a', 'перший'), ('d',
'четвертий'))]
Видаляється елемент зі значенням: другий
Словник: {'a': 'перший', 'd': 'четвертий'}
Словник: {}

```

У цьому разі ми створюємо два словники `symbols=dict([["a", "перший"], ["b", "другий"]])` і `more_symbols=dict([["c", "третій"], ["d", "четвертий"]])`, після чого командою `symbols.update(more_symbols)` розширяємо словник `symbols` за рахунок словника `more_symbols`. При цьому словник `symbols` змінюється, а словник `more_symbols` залишається таким, як і раніше.

Довжину словника (кількість ключів у словнику) визначаємо за допомогою функції `len()`. Звертання до елемента словника виконується за допомогою методу `get()`. Інструкція `symbols.get("c", "немає такого ключа!")` повертає результатом значення елемента словника `symbols` із ключем `"c"`, якщо такий ключ є, і значення `"немає такого ключа!"`, якщо ключа `"c"` у словнику `symbols` немає. Для перевірки входження ключа `"c"` у словник `symbols` використано інструкцію `"c" in symbols`: значення цього виразу дорівнює `True`, якщо ключ `"c"` є в словнику `symbols`, а якщо ключа `"c"` у словнику `symbols` немає, то значення виразу дорівнює `False`.

Щоб видалити елемент із ключем `"c"` зі словника `symbols` використовуємо команду `del symbols["c"]`. При видаленні елемента словника команда `symbols.pop("b")` не тільки видаляється елемент із ключем `"b"`, але ще й повертається значення елемента, який видаляють. Нарешті, команда `symbols.clear()` виконує повну очистку словника: з нього видаляються всі елементи.



Для створення копії словника використовують метод `copy()` і функцію `deepcopy()` із модуля `copy`. Метод `copy()` створює поверхневу копію словника. Функція `deepcopy()` створює повну копію словника. Тобто в такому разі має місце аналогія зі списками.

Так само, як для списків, для словників існують генератори. Якщо порівнювати *генератор словників* із генератором списків, то існують деякі відмінності (у принципі, їх дві):

- На відміну від генератора списку, який береться у прямокутні дужки, генератор словника береться у фігурні дужки.
- При створенні словника за допомогою генератора за кожну ітерацію (цикл) доводиться «генерувати» два

параметри: ключ і елемент. Ключ і елемент розділяються двокрапкою. Наприклад, якщо у нас є два списки `clr = ["червоне", "жовте", "зелене"]` і `txt = ["стоїмо", "чекаємо", "рухаємося"]`, то командою `A = {clr[i]:txt[i] for i in range(len(clr))}` створюється словник, у якому елементи первого списка `clr` відіграють роль ключів, а відповідні їм елементи другого списка `txt` — власне елементів словника `A`.

Далі розглянемо деякі особливості реалізації текстових значень у Python.

Текстові рядки

Мої критики не усвідомлюють, що я не роблю вербальних помилок. Я висловлююсь досконалими формами і віршованими розмірами давнього хайку.

Дж. Буш (молодший)

У наших програмних кодах ми постійно використовуємо текстові значення, хоча при цьому особливо серйозної уваги такому типу даних, як текст, не приділяли. Настав час усунути цю «неправедливість».

Про текст (або текстові рядки) можна говорити і писати дуже багато — паче що об'єкт дослідження дає для цього всі підстави. Щоб не відволікатися на другорядні питання, ми виокремимо основні, найцікавіші теми, на яких і зосередимося.

Нас цікавитимуть:

- створення текстових рядків;
- основні операції з рядками;
- використання в рядках спеціальних символів;
- форматування текстових рядків.

Але навіть у рамках цих тем ми обмежуватимемося лише найцікавішими і найпоказовішими випадками.

Текстові рядки ми вже створювали: текстове значення брали в подвійні лапки. Насправді, це не єдиний спосіб створити текстовий літерал. Із таким же успіхом можна брати текст в одинарні лапки. В які лапки брати

текст (одинарні або подвійні) — різниці немає. Що так, що інакше — результат один і той самий. Тобто якщо ми маємо справу зі звичайним текстовим рядком, що не містить якихось спеціальних символів, то вибір типу лапок — питання суто естетичних смаків програміста. Однак, якщо ми збираємося використовувати в тексті однаарні або подвійні лапки, то звичайно використовують такий прийом:

- якщо в тексті є подвійні лапки, то весь текст беруть в однаарні лапки;
- якщо в тексті є однаарні лапки, то весь текст беруть у подвійні лапки.

Ці правила зручні, але не обов'язкові. Наприклад, ми можемо в тексті, виділеному двома лапками, використовувати подвійні лапки — але в цьому разі подвійним лапкам у тексті повинна передувати обернена скісна риска (бекслеш \). Це ж зауваження справедливе й тоді, коли в тексті, виділеному однаарними лапками, потрібно використовувати апостроф.



Якщо в текстовому значенні потрібно помістити символ \ не як частину якоїсь спеціальної інструкції, а як букву (тобто якщо ми хочемо, щоб у тексті відображувався символ \), то насправді потрібно використовувати дві обернені скісні риски, тобто \\ . Іншими словами, комбінація \\ у тексті відображається як однаарна обернена скісна риска.

Корисною буде обернена скісна риска й у тому разі, якщо ми захочемо створити текстовий літерал (текстове значення), який займає декілька рядків коду. Символ \ у кінці рядка коду всередині текстового літерала означає перенос рядка.



Символ переносу \ повинен бути останнім у рядку команди — тобто всередині текстового літерала в рядку праворуч від символу \ жодних інших символів (букв) бути не повинно. Також важливо розуміти, що в цьому випадку йдеться про розміщення текстового літерала в декількох рядках у вікні редактора кодів. Під час відображення такого текстового літерала переході до нового рядка в місці розташування символу \ не відбувається. Інструкцією переходу до нового рядка є \n. Іншими словами, якщо необхідно, щоб під час відображення тексту в певному місці цього тексту виконувався переході до нового рядка, у цьому місці вставляємо інструкцію \n.

Можна вчинити ще радикальніше: взяти текстовий літерал у потрійні подвійні лапки (тобто на початку тексту тричі подвійні лапки і тричі подвійні лапки в кінці тексту) або потрійні одинарні лапки (три одинарні лапки на початку тексту і три одинарні лапки в кінці тексту). Перевага такого підходу полягає в тому, що всередині відповідного текстового значення можна вільно використовувати одинарні і подвійні лапки, а також виконувати розбивку літерала на декілька рядків. Причому при відображені літерала розбивка на рядки залишається такою, якою вона була у вікні редактора кодів. Невеликі приклади оголошення текстових значень наведено в лістингу 5.11.

Лістинг 5.11. Створення тексту

```
txt="Знання мови 'Python' - запорука успіху."  
print(txt)  
txt='Знання мови "Python" - запорука успіху.'  
print(txt)  
txt="Знання мови \"Python\" - запорука успіху."  
print(txt)  
txt='Знання мови \'Python\' - запорука успіху.'  
print(txt)  
txt="Знання мови 'Python' \  
    - запорука успіху."  
print(txt)  
txt="Знання мови 'Python'\n - запорука успіху."  
print(txt)  
txt="""Знання мови  
        "Python"  
        - запорука успіху."""  
print(txt)
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 5.11)

```
Знання мови 'Python' - запорука успіху.  
Знання мови "Python" - запорука успіху.  
Знання мови "Python" - запорука успіху.  
Знання мови 'Python' - запорука успіху.
```

Знання мови 'Python' - запорука успіху.

Знання мови 'Python'

- запорука успіху.

Знання мови

"Python"

- запорука успіху.

Хочеться вірити, що особливих коментарів і код, і результат його виконання не потребують.

Оскільки текстовий рядок є впорядкованим набором символів, то немає нічого дивного в тому, що до рядків застосовні такі операції, як звертання до елемента рядка (букви) за індексом і отримання зрізу. Як і у випадку з іншими впорядкованими множинами типів даних, таких як кортежі та списки, індексація елементів (букв) у текстовому рядку починається з нуля.

Дізнатися кількість букв (символів) у рядку можна за допомогою функції `len()`. Конкатенацію (об'єднання) рядків виконують, як ми вже знаємо, за допомогою оператора додавання `+`.



Окрім процедури об'єднання рядків звичайним «додаванням» (тобто за допомогою оператора `+`), у Python існує так звана неявна **конкатенація рядків**. У цьому випадку рядки просто розміщують один поруч із іншим, без жодного оператора між ними. Наприклад, так: `txt="Ми вивчаємо " "Python"`. Тобто ми розташували поруч (через пробіл) два літерали: "Ми вивчаємо" і "Python". У результаті змінна `txt` отримує значення "Ми вивчаємо Python".

Приклади звертання до символів текстового рядка через індекс, отримання зрізу, а також конкатенації рядків наведено в лістингу 5.12.

Лістинг 5.12. Текстові рядки

```
# Неявна конкатенація рядків
txt="Ми вивчаємо " "Python"
print(txt)
```

Phyton

```
print("Усього",len(txt),"букв")
# Використано символ табуляції \t
print("Індекс\tБуква")
# Відображаються індекси й букви
for i in range(len(txt)):
    # Перетворення (за допомогою функції str())
    # цілочислового типу у текстовий,
    # звертання до букви за індексом
    print(str(i)+": \t"+txt[i])
print(txt[12:])
# За допомогою зрізу рядок відображається
# у зворотному порядку
print(txt[::-1])
# Текстовий рядок
word="Java"
# Явна конкатенація рядків і присвоювання
# змінній txt нового значення
txt=txt[:3]+"нe"+txt[2:12]+word
print(txt)
```

Нижче наведено результат виконання цього програмного коду:

■ Результат виконання програми (з лістингу 5.12)

Ми вивчаємо Python

Усього 18 букв

Індекс Буква

0: М

1: и

2:

3: в

4: и

5: в

6: ч

7: а

8: е

9: м

10: о

```

11:
12:     P
13:     Y
14:     t
15:     h
16:     o
17:     n
Python
nohtyP омєачвив иM
Ми не вивчаемо Java
.....
```

У наведеному програмному коді, хоча він простий та очевидний, варто звернути увагу на деякі моменти. Так, вихідний текстовий рядок `txt` створюється неявною конкатенацією текстових літералів (два текстових літерали підряд без оператора між ними). Для визначення кількості букв у текстовому рядку `txt` використовуємо інструкцію `len(txt)`. У текстових літералах декілька разів використовується спеціальний символ `\t` — це символ табуляції.



Спеціальні символи, такі як `\n` (переведення рядка) або `\t` (табуляція), не відображаються, а мають деякий «таємничий зміст». Хоча формально символів два (в інструкції `\n` це `\ i \n`), розглядаються такі спеціальні символи як один. Уставляються вони прямо в текстовий рядок.

В операторі циклу звертання до букви в текстовому рядку `txt` виконується у форматі `txt[i]` (де `i` — індексна змінна). При цьому під час обчислення текстового виразу `str(i) + "`: `\t" + txt[i]`, що є конкатенацією трьох текстових фрагментів, для переведення цілочислового значення індексної змінної `i` в текстовий формат використано функцію `str()`.

Також програмний код містить приклади виконання зрізу для текстового рядка. Так, інструкція `txt[12:]` повертає частину текстового рядка `txt`, починаючи з 13-ї букви (цій букві відповідає індекс 12, оскільки індексація починається з нуля) і до кінця тексту. Вираз `txt[::-1]` — це текстовий рядок `txt`, записаний у зворотному порядку (оскільки крок приросту за індексом для отримання зрізу вказано від'ємний).

У команді `txt=txt[:3]+"нє"+txt[2:12]+word` при обчисленні правої частини виконується конкатенація таких текстових рядків:

- зір `txt[:3]` (у тексті `txt` букви від початку і до індексу 2 включно);
- текст "нє";
- зір `txt[2:12]` (букви в тексті `txt` від індексу 2 до індексу 11);
- текстове значення змінної `word` (значення "Java").

Отриманий текст як нове значення присвоюється змінній `txt`.



Текст (тип даних `str`) належить до незмінних типів даних, тому змінити текстове значення не можна. Тобто ми не можемо в текстовому значенні взяти й змінити, наприклад, якусь букву. Але ми можемо на основі тексту сформувати нове текстове значення, а потім змінній, яка посилається на вихідний текст, присвоїти нове значення. Завдяки тому, що в Python змінні посилаються на значення (а не містять їх), такий прийом можливий і часто використовується на практиці. Причому виникає ілюзія, начебто реально змінюється значення змінної. Насправді посилання (що міститься у змінній і пов'язує цю змінну з даними) перекидается з одного значення на інше. Саме таким прийомом ми скористалися вище.

Щодо методів (і функцій) для роботи з текстом, то їх дуже багато й діапазон їхнього призначення надзвичайно широкий. Наприклад, є декілька методів, призначених для керування реєстром символів у текстовому рядку:

- метод `upper()` дозволяє отримати рядок, у якому всі букви великі (верхній реєстр);
- метод `lower()` дозволяє отримати рядок, у якому всі букви малі (нижній реєстр);
- метод `capitalize()` дозволяє отримати рядок, у якому перша буква велика;
- метод `title()` дозволяє отримати рядок, у якому перша буква кожного слова велика;
- метод `swapcase()` дозволяє отримати рядок, у якому всі малі букви замінено на великі, а великі — на малі.

У всіх цих методів аргументів немає. Методи викликаються з текстового об'єкта (і текстової змінної). Вихідний текстовий рядок вони не змінюють,

а на основі цього рядка формують новий, який і повертається як результат методу. Приклади використання методів подано в лістингу 5.13.

Лістинг 5.13. Реєстр символів

```
txt="ми вивчаємо мову PYTHON"
print(txt.upper())
print(txt.lower())
print(txt.capitalize())
print(txt.title())
print(txt.swapcase())
print(txt)
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 5.13)

```
МИ ВИВЧАЄМО МОВУ PYTHON
```

Важливий клас задач пов'язаний із пошуком символів або текстових фрагментів у рядку. Наприклад:

- Методи `find()` і `index()` використовують для пошуку підрядка в текстовому рядку. Кожен із методів викликається з об'єкта текстового рядка (змінною, що посилається на рядок), а як аргумент передається підрядок, пошук якого виконується в рядку. Результатом є індекс позиції, починаючи з якої підрядок входить у рядок. Якщо рядок містить підрядок у декількох місцях, повертається індекс першого входження. Якщо в рядку немає підрядка, то метод `find()` повертає значення `-1`, а метод `index()` генерує помилку (клас `ValueError`). Методам можна також передати другий і третій числові аргументи. У цьому разі пошук підрядка здійснюється в діапазоні індексів, що визначаються цими аргументами. Аналогічним чином використовуються й методи `rfind()` і `rindex()`. Базова відмінність

- від методів `find()` і `index()` у тому, що тепер виконується пошук останнього входження підрядка в рядок.
- За допомогою методу `count()` можна підрахувати, скільки разів підрядок (аргумент методу) входить у рядок (об'єкт, із якого викликається метод). Пошук може виконуватися не по всьому рядку, а тільки по фрагменту рядка — другий і третій числові аргументи методу (якщо вони є) визначають діапазон пошуку.
 - Методи `startswith()` і `endswith()` дозволяють перевірити відповідно, чи починається і чи закінчується рядок (об'єкт, із якого викликається метод) підрядком (аргумент методу).
 - Метод `replace()` використовується для виконання заміни текстових фрагментів у текстовому рядку. Метод викликається з текстової змінної, а як аргументи йому передаються два текстових підрядки. Методом як результат повертається текстовий рядок, який отримуємо заміною у вихідному тексті (об'єкт, із якого викликається метод) підрядка — першого аргументу методу, на підрядок — другий аргумент методу. Кількість замін можна обмежити, передавши методу третій числовий аргумент (максимальна кількість замін).

Невеликий приклад використання деяких із перерахованих вище методів наведено в лістингу 5.14.

Лістинг 5.14. Пошук і заміна символів

```
txt="""В.С. Стус. "Сті років..." (уривок):  
Сті років як сконала Січ.  
Сибір. І соловецькі келії.  
І глупа облягає ніч  
Пекельний край і крик пекельний.  
Сті років мучених надій,  
І сподівань, і вір, і крові  
Синів, що за любов тавровані,  
Сті серць, як сті палахкотінь."""  
word="Сті"  
print(txt,end='\n\n')  
print("Підрядок зустрічається",txt.count(word),"рази")  
print("Перша позиція:",txt.index(word))  
print("Наступна позиція:",txt.find(word,13))
```

```
print("Остання позиція:",txt.rindex(word))
print("На початку ініціали:",txt.startswith("В.С."))
print("У кінці крапка:",txt.endswith("."),end=' \n\n')
print(txt.replace(" ","_"))
.....
```

Виконання програмного коду приводить до такого результату:

 **Результат виконання програми (з листингу 5.14)**

В.С. Стус. "Сто років..." (уривок) :

Сто років як сконала Січ.

Сибір. І соловецькі келії.

І глупа облягає ніч

Пекельний край і крик пекельний.

Сто років мучених надій,

І сподівань, і вір, і крові

Синів, що за любов тавровані,

Сто серць, як сто палахкотінь.

Підрядок зустрічається 4 рази

Перша позиція: 12

Наступна позиція: 36

Остання позиція: 225

На початку ініціали: True

У кінці крапка: True

В.С._Стус._"Сто_років..."_ (уривок) :

Сто_років_як_сконала_Січ.

Сибір._І_соловецькі_келії.

І_глупа_облягає_ніч

Пекельний_край_і_крик_пекельний.

Сто_років_мучених_надій,

І_сподівань,_і_вір,_і_крові

Синів,_що_за_любов_тавровані,

Сто_серць,_як_сто_палахкотінь.



У деяких командах із викликом функції `print()` серед аргументів є інструкції `end='\\n\\n'`. У цьому разі при виведенні тексту в консоль у кінці двічі виконується перехід до нового рядка. Тому після виконання такої команди у вікні виводу з'являється порожній рядок.

Є група методів, які дозволяють виконати перевірку вмісту текстової змінної. Скажімо, нас може цікавити питання, в якому реєстрі знаходяться букви в тексті або чи є в тексті цифри. Є група методів, які дозволяють отримати відповіді на такі нетривіальні питання. Назва кожного методу починається зі слова *is*. Кожен із методів викликається (без аргументів) із текстового рядка і повертає значення `True`, якщо умова (властивість тексту), яка перевіряється, істинна, і значення `False` — якщо вона хибна. Наприклад:

- метод `isdigit()` як значення повертає `True`, якщо текст складається тільки з цифр;
- метод `isalpha()` повертає значення `True`, якщо текст складається тільки з букв;
- метод `isalnum()` повертає значення `True`, якщо текст не містить нічого, крім букв і цифр;
- метод `islower()` повертає значення `True`, якщо текст складається тільки з малих букв;
- метод `isupper()` повертає значення `True`, якщо текст складається тільки з великих букв;
- метод `istitle()` повертає значення `True`, якщо кожне слово в тексті починається з великої букви.

Приклади застосування цих методів зібрано в лістингу 5.15.



Лістинг 5.15. Перевірка текстових значень

```
print("123".isdigit(),"12.3".isdigit())
print("abc".isalpha(),"abc123".isalpha())
print("ab12".isalnum(),"ab12\\n".isalnum())
print("ABC".isupper(),"aBc".isupper())
print("abc".islower(),"aBc".islower())
print("Ab12 Ab12".istitle(),"Ab12 AB12".istitle())
```

Результат виконання цього програмного коду такий:

 Результат виконання програми (з листингу 5.15)

```
True False
True False
True False
True False
True False
True False
```

Є її інші корисні методи, часто незамінні при роботі з текстовими значеннями. Вони коротко перераховані нижче.

- Методи `strip()`, `lstrip()` і `rstrip()` використовуються для видалення певних символів із текстового рядка — відповідно на початку й у кінці рядка, тільки на початку рядка і тільки в кінці рядка. Символи, які видаляють (у вигляді текстового рядка) передаються як аргумент методу. Якщо аргумент не вказати, за замовчуванням видаляються пробіли.
- Для розділення рядка на підрядки використовують метод `split()` (або `rsplit()`). Роздільник (у вигляді тексту), який слугує індикатором розбивки на рядки, передається як аргумент методу. Результатом методу повертається список із підрядків, на які розбивається вихідний рядок. Методом `split()` пошук підрядка-роздільника виконується зліва направо, а методом `rsplit()` — справа наліво. Якщо розбивка на підрядки виконується за символом переходу до нового рядка `\n`, можна використовувати метод `splitlines()`. Метод `partition()` виконує схожу процедуру. У рядку, з якого викликається метод, знаходиться перше входження підрядка, переданого як аргумент методу. Тобто аргумент методу — це підрядок-роздільник. І вихідний рядок наче розбивається на три частини: те, що до підрядка-роздільника, підрядок-роздільник і те, що після підрядка-роздільника. Усі ці три фрагменти повертаються у вигляді кортежу. У разі якщо підрядка-роздільника у вихідному рядку немає, кортеж (результат методу) буде містити як перший елемент вихідний рядок, а два інших елементи — порожній текст. Аналогічно використовується метод `rpartition()`, але тільки пошук підрядка-роздільника виконується з кінця вихідного рядка.
- Метод `join()` дозволяє створювати текстові рядки через об'єднання текстових підрядків, реалізованих у вигляді списку.

Список з об'єднуваними текстовими фрагментами передається як аргумент методу. Між фрагментами можна додавати текстовий роздільник — це той текст, із якого викликається метод `join()`.

Приклади використання деяких методів наведено в лістингу 5.16.

Лістинг 5.16. Деякі операції з рядками

```
txt="_*_ABC_*_abc_*_"
print(txt.lstrip("_*_"))
print(txt.rstrip("_*_"))
print(txt.strip("_*_"))
print(txt.split("*"))
print(txt.rsplit("*"))
print(txt.partition("*"))
print(txt.rpartition("*"))
print("abc \n ABC \n ***".splitlines())
print("_*_.join(["AAA", "BBB", "CCC"]))
```

Нижче наведено результат виконання цього коду:

Результат виконання програми (з лістингу 5.16)

```
ABC_*_abc_*
_*_ABC_*_abc
ABC_*_abc
['_', '_ABC_', '_abc_', '_']
['_', '_ABC_', '_abc_', '_']
('_', '*', '_ABC_*_abc_*_')
('_*_ABC_*_abc_', '*', '_')
['abc ', ' ABC ', ' ***']
AAA_*_BBB_*_CCC
```



Іноді доводиться мати справу з кодами символів. Функція `chr()` дозволяє за кодом символу відновити сам символ: як аргумент функції передається код, а результатом є символ. Зворотна процедура (визначення коду для

символу) виконується за допомогою функції `ord()`: як аргумент функції передається символ, а результатом функції є код цього символу.

Ще один важливий момент, який не можна обійти увагою, — це форматування текстових рядків. Ідеється про явне визначення способу і форми відображення текстових значень. У Python ця задача може вирішуватися по-різному. Наприклад, методи `center()`, `ljust()` і `rjust()` дозволяють здійснювати вирівнювання текстового рядка (відповідно — по центру, за лівим краєм і за правим краєм) всередині поля заданої ширини. Ширина поля передається як аргумент методу. Приклади використання цих методів наведені в лістингу 5.17.

Лістинг 5.17. Вирівнювання тексту

```
txt="ABCDEFGH"
print("*" + txt.center(20) + "*")
print("*" + txt.rjust(20) + "*")
print("*" + txt.ljust(20) + "*")
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 5.17)

```
*      ABCDEFGH      *
*          ABCDEFGH*
*ABCDEFGH      *
```

У цьому випадку текст вирівнюється всередині поля завширшки 20 символів, а для зручності на початку і в кінці текстових значень додано символи *.

Досить цікавий метод `format()`. Метод викликається з текстового рядка, у нього є аргументи, і як результат метод повертає текстове значення. Якщо абстрагуватися від деталей, то загальна схема така: беремо певний текстовий рядок, виконуємо з ним певні маніпуляції і в результаті отримуємо новий рядок. За «маніпуляції» якраз і «відповідає» метод `format()`. Маніпуляції виконуються над тим текстом, із якого викликається метод. А характер маніпуляцій визначається аргументами методу `format()`. Текстовий рядок, із якого викликається метод `format()`, може містити

спеціальні символи форматування — тобто вбудовані в текст інструкції, які мають особливий «зміст» для методу `format()` і оброблюються ним. Тому зазвичай текстовий рядок, із якого викликається метод `format()`, називають *рядком формату*. Ми будемо притримуватися саме такої термінології.

Інструкції форматування в рядку формату беруть у фігурні дужки. В найпростішому варіанті рядок формату є текстовим шаблоном, у певні місця якого вставляють текстові параметри. Місця для вставки текстових параметрів визначаються інструкціями форматування, а самі параметри — це аргументи методу `format()`. Цілочислові значення у фігурних дужках відповідають індексам аргументів методу `format()`. Індексація аргументів методу починається з нуля (першому аргументу відповідає індекс нуль). Наприклад, результатом виразу "Один — це {0}, а два — це {1}.".format("one", "two") є текст "Один — це one, а два — це two.". Інструкція {0} у рядку формату означає, що в цьому місці повинен бути вставленій перший аргумент методу `format()` (текст "one"). Інструкція {1} визначає місце вставки другого аргументу методу `format()` (текст "two").

Інструкції формату можуть бути більш вигадливими, ніж число у фігурних дужках, і містити більше інформації, ніж просто індекс аргументу для вставки в текстовий шаблон. Через двокрапку після індексу аргументу можна вказати мінімальну ширину поля (ціле число), яка виділяється для цього аргументу, а також спосіб вирівнювання тексту в цьому полі. Спосіб вирівнювання визначається спеціальними символами: < означає вирівнювання за лівим краєм, > — вирівнювання за правим краєм, а ^ — вирівнювання по центру. Наприклад, інструкція {0:>20} означає, що у відповідному місці потрібно вставить перший (по порядку) аргумент методу `format()`, під цей аргумент потрібно виділити поле завширшки не менше 20 символів, а текст, що вставляється, повинен вирівнюватися за правим краєм.



Числовий параметр після двокрапки визначає мінімальну ширину поля. Якщо текст, який вставляють, займає більше місця, то цей параметр ігнорується.

За замовчуванням, використовується вирівнювання тексту за лівим краєм.

Невеликий приклад використання методу `format()` наведено в лістингу 5.18.

Лістинг 5.18. Рядок формату

```
txt="{0} по {0} - буде {1}"
print(txt.format("два", "четири"))
print(txt.format("три", "дев'ять"))
print("Текст '{0}': {0:<20}.".format("abcdef"))
print("Текст '{0}': {0:^20}.".format("abcdef"))
print("Текст '{0}': {0:>20}.".format("abcdef"))
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 5.18)

```
два по два - буде четири
три по три - буде дев'ять
Текст 'abcdef': abcdef
Текст 'abcdef':      abcdef
Текст 'abcdef':          abcdef.
```

Існують й інші прийоми виконання форматування, в тому числі, і за допомогою методу `format()`. У разі необхідності читач може звернутися до спеціальної довідкової літератури або вбудованої довідки середовища розробки Python.

Резюме

Іноді люди говорять, що моя політика набагато важливіша від країни.

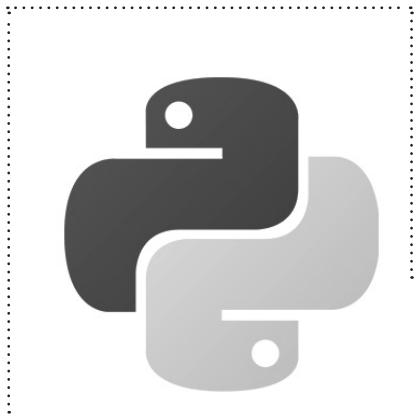
Дж. Буш (молодший)

- Множина — це невпорядкований набір унікальних елементів. Створюють множину за допомогою функції `set()` (як аргумент передається список із елементами множини) або через перерахування елементів у фігурних дужках.
- Для перевірки того, чи входить елемент у множину, використовують оператор `in`.
- Кількість елементів у множині визначають за допомогою функції `len()`.
- Для створення копії множини використовують метод `copy()`.
- Для додавання елемента в множину використовують метод `add()`, для видалення — метод `remove()`. Для додавання в множину елементів іншої множини використовують метод `update()`.
- Об'єднання множин — множина, елементи якої складаються з елементів об'єднуваних множин. Для об'єднання множин використовують метод `union()` або оператор `|`.
- Перетин множин — множина, елементи якої складаються з елементів, спільних для обох множин. Для обчислення перетину множин використовують метод `intersection()` або оператор `&`.

- Різниця множин — множина, елементи якої складаються з елементів першої множини за винятком тих елементів, які входять у другу множину. Для обчислення різниці множин використовують метод `difference()` або оператор `-`.
- Симетрична різниця множин — множина, елементи якої складаються з елементів першої і другої множин за винятком тих елементів, які входять в обидві множини одночасно. Для обчислення симетричної різниці множин використовують метод `symmetric_difference()` або оператор `^`.
- Для порівняння множин використовують оператори `==` (рівність множин), `<` (перша множина є підмножиною другої множини), `<=` (перша множина є підмножиною другої множини або множини рівні), `>` (друга множина є підмножиною першої множини), `>=` (друга множина є підмножиною першої множини або множини рівні).
- Генератор множини подібний до генератора списку, але вся конструкція береться у фігурні дужки.
- Словник є невпорядкованим набором елементів. Доступ до елементів словника виконується за ключем.
- Для створення словника використовують функцію `dict()`, аргументом якій передають список з елементами-списками. Кожен такий внутрішній список складається з двох елементів: значення ключа і значення елемента. Можна також створити словник, узявши у фігурні дужки, розділені двокрапкою, ключі й значення відповідних елементів.
- Доступ до елементів словника виконується за ключем: після імені словника у квадратних дужках указують ключ елемента.
- Метод `keys()` дозволяє отримати доступ до ключів словника. Для доступу до значень словника використовують метод `values()`. Метод `items()` повертає кортежі з ключами і значеннями елементів словника.
- Для видалення елемента зі словника застосовують метод `pop()`. Для додавання декількох елементів використовують метод `update()`. Новий елемент у словник можна додати, присвоївши значення елементу з новим ключем: після імені

словника у квадратних дужках указується ключ елемента, який додають, і після оператора присвоювання (знака рівності) — значення елемента.

- Поверхневу копію словника створюють за допомогою методу `copy()`, а повну копію можна створити за допомогою функції `deepcopy()` з модуля `copy`.
- Генератор словника беруть у фігурні дужки й одночасно створюють два параметри: ключ і відповідний йому елемент словника.
- Текстовий рядок — упорядкований набір символів. Текстові літерали беруть в одинарні або подвійні лапки.
- Для додавання апострофа, подвійних лапок, оберненої скісної риски перед цими символами вказують обернену скісну риску `\`. Обернену скісну риску використовують і в ряді спеціальних символів, таких як табуляція `\t` або інструкція переходу до нового рядка `\n`.
- До елементів (symbolів) рядка можна звертатися за індексом (індексація букв у рядку починається з нуля): після текстової змінної у квадратних дужках зазначають індекс букви в рядку. Також допускається виконання зрізів для текстового рядка.
- Змінити текстове значення не можна (але можна текстовій змінній присвоїти нове значення).
- Для конкатенації (об'єднання) текстових рядків використовують оператор `+`. Явне зведення до текстового типу виконують за допомогою функції `str()`.
- Кількість букв у текстовому рядку можна визначити за допомогою функції `len()`.
- Існує значна кількість методів, які дозволяють виконувати найрізноманітніші операції з текстовими значеннями, включаючи такі: перетворення реестру символів, пошук символів і підрядків, заміна текстових фрагментів, розбивка рядків на підрядки, перевірка вмісту текстового рядка, форматування текстового рядка для виведення в консоль і багато іншого.



РОЗДІЛ 6

**Основи об'єктно-
орієнтованого
програмування**

Віллі, ти мене налякав – тобі треба
носити дзвіночок.

З телесеріалу «Альф»

У цьому розділі ми познайомимося з основами об'єктно-орієнтованого програмування (скорочено ООП), а точніше, із тим, як принципи ООП реалізують у мові Python. В основі всієї концепції ООП, без перевільшення, є такі поняття, як *клас* і *об'єкт* (щодо Python це *клас* і *екземпляр класу*). Тому розпочнімо зі знайомства саме з цими поняттями або їхніми аналогами (хоча, безумовно, лише ними справа не обмежиться).

Класи, об'єкти й екземпляри класів

Підробка – найширіша форма плагіату.

З телесеріалу «Альф»

Перш ніж поблизу познайомитися з класами й об'єктами, наголосимо на декількох принципових положеннях, які в певному розумінні підготують читача до того підходу, який буде використано в цьому розділі для пояснення фундаментальних понять і принципів ООП. Для нас важливо ось що: їтиметься не просто про класи й об'єкти, а про те, як концепція класів й об'єктів реалізується в мові Python. Чому це важливо? Тому що сама по собі тема ООП і, конкретніше, класів і об'єктів, зазвичай досить складна для розуміння навіть для тих, хто має досвід програмування. А у випадку з мовою Python проблеми, швидше за все, виникнуть не тільки у новачків, а й у читачів, знайомих із методами ООП на прикладі таких мов, як C++, Java або C#.



Зрозуміло, що вище висловлено лише побоювання. Сподіваємося, що для читача все, що обговорюватиметься далі, буде зрозумілим або навіть очевидним. Однак статистика і практика викладання говорять про інше. Ми також виходитимемо із суворих реалій ООП. При цьому докладемо всіх зусиль, щоб вивчення азів (і не лише азів) ООП було найбільш зручним і найменш трудомістким.

Річ у тім, що механізми й принципи реалізації ООП у Python значно відрізняються від тих, що використовуються в C++, Java і C#. Принципові відмінності є навіть на рівні термінології. Тому перед нами стоїть подвійна задача: із одного боку, необхідно в доступній формі викласти

принципи ООП для тих, хто з ними не знайомий, а з іншого боку, важливо повернути «на шлях істинний» тих, хто вивчав ООП у контексті інших мов програмування.



Для тих, хто знайомий з іншими об'єктно-орієнтованими мовами: у Python клас сам є об'єктом. Ця обставина має велими серйозні наслідки. Більше того, як ми вже знаємо, змінні в Python не оголошують, а вводять в програму через присвоювання значення. Це ж правило залишається справедливим і під час роботи з класами й об'єктами. Звідси виходить, що процедура оголошення полів, стандартна для багатьох мов програмування, у Python просто втрачає зміст. Аналогічно, багато звичних (за мовами програмування C++, Java і C#) в ООП моментів виявляється чужими для мови Python. Отже, екзотики не бракуватиме.

Головна ідея ООП полягає в тому, щоб об'єднати в одне ціле дані (тобто те, що зберігається в змінних) і функції, призначенні для обробки цих даних. Реалізується ця ідея в класі. За великим рахунком, *клас* — це деяка конструкція, яка пов'язує або об'єднує певну кількість змінних і певну кількість функцій. На основі класу створюють *об'єкти*. Загалом, про клас зручно думати як про деякий шаблон, на основі якого потім «за образом і подобою» створюють об'єкти. Але важливо розуміти, що тут ідеться не про створення клонів. Тобто клас і об'єкт, створений на основі цього класу, — зовсім різні речі (або сутності). Скористаємося аналогією: припустімо, нам треба зробити (зібрати) автомобіль. Завод у нас є, і ми можемо виготовити будь-яку деталь. Але цього замало. Нам потрібен деякий план (або креслення), який би давав нам чітке й однозначне уявлення, які деталі, як і куди кріпляться в автомобілі. Зрозуміло, що насправді там дуже багато креслень для різних блоків і механізмів. Але наразі це не важливо. Ми можемо думати, що креслення одне. Ось це креслення, на основі якого збирають автомобіль, і є аналогом класу. А автомобіль, який збирається відповідно до креслення, — аналог об'єкта, створеного на основі класу.

Яку роль відіграє креслення? У ньому «прописано», які деталі є в автомобілі й що цей автомобіль «уміє робити»: скільки дверей, скільки фар, і як вони розташовані, розмір коліс, наявність або відсутність кондиціонера, радіоприймача, паркувального пристрою і багато іншого. Усі автомобілі, зібрані за цим кресленням, будуть однотипними в сенсі набору

опцій і функціональних можливостей. Якщо ми візьмемо інше креслення й зберемо за ним автомобіль, то, очевидно, що отримаємо автомобіль з дещо іншими характеристиками (усе залежить від креслення — що в ньому закладено, те й отримаємо).

Тобто ситуація така:

- є креслення автомобіля — аналог класу;
- на основі креслення можна збирати автомобілі; автомобіль, зібраний на основі креслення, — аналог об'єкта, який створено на основі класу.

Але це ще не все. Адже креслення повинно бути «записане» на якомусь «носії». Сьогодні, звичайно, «носіями» є комп'ютери, але нам зручніше думати, що креслення виконано на аркуші паперу. Аркуш паперу з кресленням — це теж об'єкт, але інший, не такий, як автомобіль. Тобто насправді, є такий об'єкт, як креслення. На основі цього об'єкта можна створити інший об'єкт — автомобіль. Тут «приховано» важливу обставину, специфічну саме для мови Python: у цій мові об'єктом є не тільки те, що ми створюємо на основі класу, але й сам клас є об'єктом. Так само, як паперовий аркуш із кресленням є об'єктом, відмінним від автомобіля, об'єкт, яким є клас, відрізняється від об'єкта, який створюється на основі класу. І тут ми підходимо до дуже важливого місця: до *термінології*. Той об'єкт, який створюється на основі класу, ми будемо називати *екземпляром класу*. Той об'єкт, через який реалізується клас, будемо називати *об'єктом класу*.



Для фанатів C++, Java і C# такий термінологічний підхід може здатися свавіллям (річ у тому, що в цих мовах об'єктом класу зазвичай називають той об'єкт, що створюється на основі класу). Але ми так робимо через необхідність і дотримуємося тієї традиції іменування класів і об'єктів, яка «історично» склалася серед розробників мови Python і програмістів, що використовують цю мову.

Отже, далі ми будемо називати об'єкти, які створюються на основі класу, *екземплярами класу* або просто *екземплярами*. Також, у деяких випадках, використовують термін *об'єкт-екземпляр*. Те, що клас у Python насправді є об'єктом — обставина важлива, але зараз ми на цьому не акцентуємо.

Хоча потім і згадаємо про таку його особливість. Тепер сконцентруємося на двох запитаннях:

- як створити (описати) клас;
- як на основі класу створити екземпляр класу?

Відповіді на ці запитання одночасно прості й складні. Вони прості, якщо дивитися на все це з формальної точки зору. А складними вони стають, щойно ми починаємо «копати вглиб».

Отже, шаблон опису класу має такий вигляд (жирним шрифтом виділено ключові інструкції):

```
class ім'я_класу:  
    # тіло класу
```

Починається все з ключового слова `class`, після якого вказують *ім'я класу* і двокрапку. Потім описують тіло класу (нагадуємо, що під час опису інструкцій у тілі класу необхідно робити відступи — рекомендовано використовувати чотири пробіли). Що ж пишуть у тілі класу? У тілі класу, як правило, описують *методи*. Метод — це та ж функція, тільки викликатиметься вона з екземпляра класу. Ми з методами вже мали справу неодноразово. Але раніше ми використовували готові методи, а тепер нам доведеться все це організувати своїми руками.

Перш ніж розглянути конкретний приклад, відзначимо одну важливу обставину. Формально метод у тілі класу описують як звичайну функцію. Але методи, як ми вже знаємо, повинні викликатися з екземпляра класу. В описі методу екземпляр класу, із якого буде викликано метод, повинен бути явно вказаній як перший аргумент методу. Під час виклику методу цей аргумент методу явно не передається. Причина в тому, що екземпляр класу, з якого викликають метод, зазначають явно (перед іменем методу через крапку). Виходить таке своєрідне правило, яке умовно можна назвати «мінус один аргумент»: під час виклику методу з екземпляра класу в нього на один аргумент менше, ніж це було під час опису методу (це якщо немає аргументів зі значенням за замовчуванням).



Іншими словами, під час опису методу в тілі класу в нього повинен бути принаймні один аргумент. Перший аргумент у списку аргументів методу позначає той екземпляр класу, з якого викликають (або якщо точніше, викликатимуть) метод. Коли метод викликають, то найперший із аргументів йому не передається (точніше, у круглих дужках після імені методу не вказується) — як перший аргумент автоматично використовується посилання на екземпляр класу, з якого викликано метод.

Ну і, звісно, із того, що у методу є аргумент, ще не випливає, що цей аргумент обов'язково повинен бути задіяний у програмному коді методу.

Існує угода називати під час опису методу перший його аргумент `self`. Ми також будемо дотримуватися цієї угоди — тобто в тих місцях програмного коду, де у методу оголошено аргумент із назвою `self`, цей аргумент означатиме посилання на екземпляр класу, з якого викликається метод.

Після того, як клас створено, виникає наступне запитання: як на основі класу створити екземпляр класу? Робиться це дуже просто за допомогою команди вигляду `змінна=клас()`. Іншими словами, після імені класу ставимо круглі дужки (поки що порожні) і всю цю конструкцію присвоюємо як значення деякій змінній. Як результат буде створено екземпляр класу, а посилання на цей екземпляр записано в змінні.



Команда створення екземпляра класу може бути більш хитромудрою. Але цю ситуацію ми обговоримо після того, як познайомимося з конструкторами.

Метод, описаний у класі як метод екземпляра класу (а ми поки інших варіантів не знаємо), викликається з екземпляра класу за допомогою «крапкового синтаксису»: після екземпляра класу через крапку вказують ім'я методу, який викликають. При цьому методу в круглих дужках передаються потрібні аргументи. Якщо аргументи методу передавати не треба, порожні круглі дужки після імені методу все одно ставимо. Невеликий приклад створення класу й екземпляра класу (з наступним викликом із екземпляра класу методу) наведено в лістингу 6.1.

Лістинг 6.1. Клас і екземпляри класу

```
# Створюємо клас із назвою MyClass
class MyClass:
    # Метод екземпляра класу.
    # Єдиний аргумент методу self – посилання
    # на екземпляр класу, з якого викликається
    # метод
    def say_hello(self):
        # Метод відображує повідомлення. Аргумент
        # методу (посилання self) явно не використовується
        print("Вас вітає екземпляр класу!")

# Створюємо екземпляр класу
obj=MyClass()
# Викликаємо метод екземпляра класу.
# Під час виклику аргументи методу не передаються
obj.say_hello()
```

У результаті виконання цього програмного коду у вікні виводу з'являється таке повідомлення:

Результат виконання програми (з лістингу 6.1)

Вас вітає екземпляр класу!

У цьому разі ми створюємо клас із назвою `MyClass`. У тілі класу описано всього один метод, який називається `say_hello()`, і у цього методу, як нескладно помітити, один аргумент, який називається `self`. У тілі методу є одна команда `print("Вас вітає екземпляр класу!")`, після виконання якої, як ми небезпідставно сподіваємося, у вікні виводу з'явиться текстове повідомлення.



Хоча метод `say_hello()` оголошено з аргументом `self`, цей аргумент у тілі методу насправді не використовується. Тут нема нічого «незаконного». Хорошого, правда, також мало. Коментарі – далі.

Екземпляр класу створюється командою `obj=MyClass()`. У цьому випадку його створено безпосередньо під час виконання інструкції `MyClass()`, а посилання на цей екземпляр записується в змінну `obj`. Але ми, якщо це не викликатимемо непорозумінь, тут і далі будемо називати екземпляром класу змінну, яка насправді всього лише посилається на екземпляр класу. Метод `say_hello()` із екземпляра класу (zmінна `obj`) викликають командою `obj.say_hello()`. У принципі, такого роду команди, коли метод викликають з екземпляра класу, ми вже використовували багато разів. Звертають на себе увагу дві обставини. По-перше, хоча метод `say_hello()` описувався в класі з одним аргументом, викликається він без передачі аргументів. Тобто отримуємо правило «мінус один аргумент» у дії. По-друге, результат виконання методу `say_hello()` унаслідок того, як його визначено, не залежить від екземпляра класу, з якого викликають метод. Якщо б ми на основі класу `MyClass` створили ще один екземпляр класу й викликали з нього метод `say_hello()`, результат був би таким самим, як і в розглянутому вище прикладі. Хоча нічого неправильного або некоректного в цьому немає, такий підхід межує з недоречним стилем. Чому? Як мінімум тому, що якщо нас влаштовує ситуація, коли всі екземпляри класу «поводяться» однаково, то виникають сумніви в необхідності використання ООП для розв'язання відповідної задачі. Простіше кажучи, тут має місце не зовсім адекватне застосування об'єктно-орієнтованого підходу. Тому зазвичай методи, які описуються як методи екземпляра класу, у тому або іншому вигляді використовують посилання на екземпляр класу, з якого викликається метод (перший аргумент із рекомендованою назвою `self`).

Перш ніж розпочати розгляд прикладу, в якому нам пощастиТЬ уникнути описаної вище прикрої неприємності, розширимо пізнання про вміст екземплярів класу. Новина дуже проста: в екземплярів класу можуть бути не лише методи, а й змінні, які ми будемо називати *полями екземпляра класу*. А все разом, — поля й методи екземпляра класу, — називатимемо *атрибуутами екземпляра класу*. Тобто атрибути — це поля й методи.



Ситуація з термінологією не є однозначною. Термін поле швидше стосується таких мов, як C++, Java або C#. У довідкових ресурсах з мовою Python те, що ми будемо називати полем, зазвичай називають дещо інакше: *атрибуути-дані, змінні екземпляра класу*, рідше — *властивість*, а іноді просто

використовують термін **атрибут**. Втім, термін **поле** досить компактний і зручний. Ним і будемо послуговуватися.

Фактично поле екземпляра класу — це деяка змінна, яка «приписана» (або «прикріплена») до цього екземпляра класу. Проблема тут ось у чому: змінні, як ми знаємо, з'являються тоді, коли їм присвоюють значення. Якщо ми говоримо про змінну в контексті екземпляра класу, то природно виникає запитання: коли і як змінній (полю) можна присвоїти значення? Логічно припустити, що під час виконання методу екземпляра класу. Невеликий приклад, у якому реалізується такий підхід, наведено в лістингу 6.2.

Лістинг 6.2. Поле екземпляра класу

```
# Створюємо клас
class MyClass:
    # Метод для присвоювання значення
    # полю екземпляра класу
    def set(self, n):
        print("Увага! Присвоюється значення!")
        # Полю присвоюється значення
        self.number=n
    # Метод для зчитування значення
    # поля екземпляра класу
    def get(self):
        # Відображаємо значення поля
        print("Значення поля:", self.number)
# Створюємо екземпляр класу
obj=MyClass()
# Викликається метод екземпляра класу
# й полю екземпляра класу присвоюється
# значення
obj.set(100)
# Викликається метод екземпляра класу
# й відображується значення поля екземпляра
# класу
obj.get()
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

Після виконання цього програмного коду у вікні виводу з'являються два повідомлення, як показано нижче:



Результат виконання програми (з лістингу 6.2)

Увага! Присвоюється значення!

Значення поля: 100

Проаналізуємо програмний код. Ми, як і в попередньому прикладі, створюємо клас із назвою `MyClass`. Але тепер у цьому класі описано два методи екземпляра класу: метод `set()` призначений для присвоювання значення полю екземпляра класу. Поле називається `number`, і про його існування ми дізнаємося виключно з програмного коду методу `set()`: у тому місці, де командою `self.number=n` полю `number` екземпляра класу, на який посилається змінна `self`, присвоюється значення змінної `n`. І змінна `self` (перший аргумент), і змінна `n` (другий аргумент) оголошені як аргументи методу `set()`. Змінна `self` є посиланням на екземпляр класу, з якого викликається метод, а змінна `n` — це безпосередньо той аргумент, який передається методу під час виклику. Пізніше, коли командою `obj=MyClass()` буде створено екземпляр `obj` класу `MyClass` і потім командою `obj.set(100)` із екземпляра `obj` викликано метод `set()` з аргументом `100`, це значення буде присвоєно полю `number` екземпляра `obj`.



Перед тим, як метод `set()` полю `number` присвоює значення, командою `print("Увага! Присвоюється значення!")` у тілі методу виводиться повідомлення. Тож процес присвоювання значення полю непоміченим не проходить.

Метод екземпляра класу `get()` призначений для зчитування значення поля `number` екземпляра класу (точніше, для відображення значення цього поля у вікні виводу). У тілі методу всього одна команда `print("Значення поля:", self.number)`, якою й вирішується поставлена перед методом задача. Посилання `self.number` на поле `number` екземпляра класу виконується через змінну `self` — єдиний аргумент методу екземпляра класу. Під час виклику з екземпляра класу методу `get()` аргументи не передаються. Так, у результаті виконання команди `obj.get()` у вікні виводу відображується значення поля `number`.

екземпляра `obj`. Важливо те, що команда `obj.get()` виконується після команди `obj.set(100)`, оскільки, перш ніж значення поля отримати, це значення полю треба присвоїти.



Насправді, ситуація ще «жорсткіша»: під час першого виклику методу `set()` з екземпляра класу полю `number` не просто присвоюється значення — це поле створюється. Тобто доки не виконано команду присвоювання значення полю `number` екземпляра `obj` (а в цьому випадку таке присвоювання виконується методом `set()`), в екземпляра `obj` поля `number` нібито й не існує. Узагалі.

Значення полю екземпляра класу можна присвоїти через пряме звертання до цього поля. Розгляньмо невеликий приклад у лістингу 6.3.



Лістинг 6.3. Значення поля екземпляра класу

```
# Створюємо клас без методів
class MyClass:
    pass
# Створюємо екземпляр класу
obj=MyClass()
# Присвоюється значення полю number
# екземпляра obj
obj.number=100
# Відображується значення поля number
# екземпляра obj
print("Значення поля:", obj.number)
```

Результат виконання програмного коду такий:



Результат виконання програми (з лістингу 6.3)

```
Значення поля: 100
```

У цьому разі ми створюємо клас, у якому взагалі нічого немає — жодні методи в тілі класу `MyClass` не описуються. Там є тільки інструкція `pass`. За допомогою цієї інструкції ми виділяємо тіло класу. Проблема в тому, що якщо там не написати взагалі нічого, то такий синтаксис міститиме

Розділ 6 . Основи об'єктно-орієнтованого програмування

помилку. Тому, навіть якщо клас нічого не містить, щось там все одно повинно бути. У таких випадках використовуємо формальну інструкцію `pass`.

Екземпляр класу, як і в усіх попередніх випадках, створюємо командою `obj=MyClass()`. Потім за допомогою команди `obj.number=100` екземпляр `obj` отримує поле `number`, і цьому полю присвоюється значення 100. Потім значення поля `number` екземпляра `obj` зчитується й відображується у вікні виводу командою `print ("Значення поля:", obj.number)`.



Не обов'язково бути Шерлоком Холмсом, щоб зрозуміти: у Python різні екземпляри одного й того ж класу можуть мати різний набір полів. Такий стан речей є абсолютно нереальним у C++, Java, C#: у цих мовах програмування повний набір характеристик екземплярів класу (за нашою термінологією) визначається раз і назавжди тим, що описано в класі, на основі якого створюються екземпляри. Тому важливо розуміти, що класи й екземпляри класу в мові Python – це далеко не одне й те саме, що класи й об'єкти в мовах C++, Java і C#.

Ситуація, коли значення полям екземплярів класу присвоюють (явно або за допомогою спеціальних методів) уже після створення екземпляра класу не дуже зручна. Особливо вона незручна, якщо доводиться мати справу зі значною кількістю екземплярів. Існує механізм, який дозволяє частково (або навіть повністю) зняти цю проблему. Йдеться про використання *конструктора*.

Конструктор і деструктор екземпляра класу

— Альф, чого ми тебе терпимо?
— Бо я сонце вашого життя.

З телесеріалу «Альф»

У класі можна описати спеціальний метод, який називається *конструктор екземпляра класу* (іноді також називають *методом ініціалізації*). Цей метод автоматично викликається при створенні екземпляра класу. Рецепт створення конструктора дуже простий: необхідно описати метод із назвою `__init__()` (два символи підкреслювання на початку і два символи підкреслювання в кінці). Аргументів у конструктора може бути скільки завгодно — але не менше, ніж один (посилання на екземпляр, при створенні якого викликається конструктор).



У Python є група спеціальних методів, які дозволяють створювати гнучкі й ефективні програмні коди. Назва таких методів починається з подвійного підкреслювання і закінчується подвійним підкреслюванням. Із багатьма з цих методів і їхнім призначенням ми ще познайомимося. У цьому випадку ми обговорюємо конструктор `__init__()`.

У лістингу 6.4 наведено приклад створення класу, який містить конструктор.

Лістинг 6.4. Конструктор екземпляра класу

```
# Створюємо клас
class MyClass:
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

```
# Конструктор
def __init__(self):
    # Присвоюється значення полю
    self.number=0
    # Відображується повідомлення
    print("Створено екземпляр класу!")

# Створюється екземпляр класу
obj=MyClass()
# Перевіряємо значення поля екземпляра класу
print("Значення поля:", obj.number)
```

Результат виконання програмного коду такий:

■ Результат виконання програми (з листингу 6.4)

Створено екземпляр класу!

Значення поля: 0

У класі MyClass описано конструктор `__init__()` із одним аргументом `self` (у випадку конструктора це посилання на створюваний екземпляр класу). У тілі конструктора командою `self.number=0` полю `number` екземпляра класу присвоюється нульове значення, а потім командою `print("Створено екземпляр класу!")` у вікні виводу відображується повідомлення. Ці дві дії запрограмовано в конструкторі, який, нагадаємо, автоматично викликається при створенні екземпляра класу. Тому кожного разу, коли створюється екземпляр, до цього екземпляра автоматично додаватиметься поле `number` із нульовим значенням і після цього, знову ж автоматично, з'являтиметься повідомлення у вікні виводу. Тому при створенні екземпляра `obj` командою `obj=MyClass()` у цього екземпляра з'являється поле `number`, і цьому полю присвоюється значення 0. Також у вікні виводу з'явиться повідомлення Створено екземпляр класу!. За допомогою команди `print("Значення поля:", obj.number)` ми перевіряємо, чи справедливі всі ті твердження щодо поля `number` екземпляра `obj`, які було зроблено вище. Результат виконання цієї команди не дає підстав для сумнівів.

У конструктора може бути декілька аргументів (у всякому разі, більше за один). У такому випадку під час створення екземпляра класу

конструктору треба передати необхідні аргументи. Як це робиться, проілюстровано листингом 6.5.



Листинг 6.5. Аргументи конструктора

```
# Створюємо клас
class MyClass:
    # Метод для присвоювання значення полю
    def set(self, n):
        # Полю number присвоюється значення
        self.num=n
    # Метод для відображення значення поля
    def get(self):
        # Відображується значення поля number
        print("Значення поля:",self.num)
    # Конструктор із двома аргументами.
    # У другого аргументу є значення
    # за замовчуванням
    def __init__(self,n=0):
        # Викликається метод set() для присвоювання
        # значення полю number
        self.set(n)
        # Відображується повідомлення
        print("Створено екземпляр класу.")
        # Викликається метод get() для відображення
        # значення поля number
        self.get()
# Створюється екземпляр класу
a=MyClass()
# Створюється ще один екземпляр класу
b=MyClass(100)
```

Тут ми створюємо клас `MyClass`, у тілі якого описано методи екземпляра класу `set()` і `get()`. Перший метод призначений для присвоювання значення полю `number` екземпляра класу, а другий метод потрібен для відображення значення цього поля. У конструктора тепер два аргументи. Перший, як завжди, є посиланням на екземпляр класу, а другий аргумент, за нашим задумом, визначає значення поля `number` екземпляра

Розділ 6 . Основи об'єктно-орієнтованого програмування

класу. Причому цей другий аргумент має нульове значення за замовчуванням: тобто, якщо під час створення екземпляра аргумент указати, то це буде значення поля `number`, а якщо аргумент не вказати, то у поля `number` буде нульове значення. Виникає запитання: а як передати аргумент конструктору? Дуже просто: у команді створення екземпляра класу після імені класу в круглих дужках (які до цього в нас завжди були порожніми) передаються аргументи конструктору. Іншими словами, шаблон команди створення екземпляра класу з передачею аргументів конструктору такий: `змінна=клас(аргументи)`. Щодо цього конкретного випадку, то оскільки другий аргумент має значення за замовчуванням, ми можемо як передавати аргумент конструктору, так і не передавати. Прикладом першої ситуації є команда `b=MyClass(100)` (екземпляр `b` створюється зі значенням 100 для поля `number`), а другої — команда `a=MyClass()` (екземпляр `a` створюється зі значенням 0 для поля `number`).

Результат виконання наведеного вище коду такий:

■ Результат виконання програми (з лістингу 6.5)

Створено екземпляр класу.

Значення поля: 0

Створено екземпляр класу.

Значення поля: 100

Усі повідомлення відображуються під час виконання програмного коду конструкторів, оскільки програма, крім створення класу та двох екземплярів класу, жодних інших командних блоків не містить.

Метод із назвою `__del__()` (два символи підкреслювання на початку і два — в кінці) автоматично викликається під час видалення екземпляра класу з пам'яті. Цей метод прийнято називати *деструктором*. У деструктора один і тільки один аргумент (посилання на екземпляр класу `self`). Невеликий приклад із використанням деструктора наведено в лістингу 6.6.

■ Лістинг 6.6. Деструктор екземпляра класу

```
# Клас із конструктором і деструктором
class MyClass:
```

Phyton

```
# Конструктор
def __init__(self):
    print("Вітаю!")
# Деструктор
def __del__(self):
    print("Бувайте!")
print("Перевіряємо роботу деструктора.")
# Створюємо екземпляр класу
obj=MyClass()
print("Екземпляр класу створено. Видаляємо його.")
# Видаляємо екземпляр класу
del obj
print("Виконання програми закінчено.")
```

Ми створюємо клас, у якому описано конструктор і деструктор. І в тілі конструктора, і в тілі деструктора виконується по одній команді. І під час виклику конструктора, і під час виклику деструктора у вікні виводу відображуються повідомлення — тільки різні для конструктора і деструктора. Під час створення екземпляра класу з'явиться повідомлення Вітаю!, а під час видалення екземпляра класу з пам'яті з'являється повідомлення Бувайте!. Щоб видалити екземпляр класу (який до цього був створений командою `obj=MyClass()`), використовуємо інструкцію `del obj`. Можливий результат виконання програмного коду наведено нижче:



Результат виконання програми (з лістингу 6.6)

Перевіряємо роботу деструктора.

Вітаю!

Екземпляр класу створено. Видаляємо його.

Бувайте!

Виконання програми закінчено.

Хоча вся ця схема з використанням деструктора виглядає доволі елегантно, у неї є суттєвий недолік, причому стосується він не тільки розглянутого прикладу, але деструкторів узагалі. Річ у тім, що очистка пам'яті в Python виконується автоматично. Видаляються об'екти, на які в програмі немає посилань. Але сказати, коли конкретно вони будуть видалені, практично неможливо. Тому якщо деякий об'ект має бути видаленим,

Розділ 6 . Основи об'єктно-орієнтованого програмування

то його рано чи пізно буде видалено. Але коли саме — питання складне. У контексті використання деструкторів для екземплярів класу це означає, що ми можемо стверджувати, що деструктор буде викликано (тоді, коли екземпляр реально видаляється з пам'яті), але не зовсім точно можемо сказати, коли саме це відбудеться. Тому деструктори в Python використовують не дуже часто.

Поле об'єкта класу

Добре, Віллі, але не замикайте двері: у мене вночі можуть виникнути ще політичні питання.

З телесеріалу «Альф»

Зараз саме час пригадати, що клас сам по собі є об'єктом. Саме так, як реальним об'єктом є аркуш паперу, на якому виконано креслення автомобіля. Досі ми, образно висловлюючись, мали справу з побудовою «автомобілів». Тепер уважніше подивимося на те «креслення», з якого все почалося.

А ми почнемо з важливого моменту: програмний код класу *виконується*. До цього ми в тілі класу розташовували тільки оголошення методів, і жодних інших цікавих команд там не було, хоча теоретично вони там можливі (інше питання, чи вони там потрібні). Найпростіше, що спадає на думку, — розмістити в тілі класу команду присвоювання значення змінній. У такому разі отримаємо поле, але не екземпляра класу, а безпосередньо самого класу (або *поле об'єкта класу*).



Про поле об'єкта класу можна думати як про змінну, яка «відома» тільки в тілі класу. За межами класу доступ до такої змінної здійснюється через явне заначення класу, в якому її оголошено. Використовується «крапковий синтаксис» вигляду **клас.змінна**.

Якщо повернутися до аналогії з кресленням на аркуші паперу, то поле об'єкта класу — це такий собі стікер з нагадуванням або якоюсь інформацією, приклесній до аркуша паперу. Усі, хто користується кресленням, бачать і цей стікер. Але на процес створення автомобілів за кресленням

він не впливає. Це просто такий «додаток» до важливого документа, якась додаткова інформація.

З точки зору прикладного програмування, на цей момент для нас важливі дві обставини:

- у тілі класу можна присвоїти значення змінній (поле класу);
- звертатися до такої змінної слід так: указати ім'я класу і через крапку ім'я змінної.



Нерідко в довідковій літературі можна зустріти думку, що поля класу відіграють роль статичних змінних (як у мовах C++, Java, C#). Для тих, хто не знайомий із цим терміном: статична змінна або статичне поле — це поле, спільне для всіх екземплярів класу. Звичайно, у певному сенсі можна ставитися до полів класу, як до статичних змінних. Але насправді, це не зовсім так — тобто, повної аналогії немає. З одного боку, ми можемо, за певних обставин (які обговорюються далі) і тільки для зчитування значення, звертатися до поля класу через екземпляр класу. У такому разі замість імені класу вказується ім'я екземпляра класу (але й тут не все так просто). Щодо присвоювання значення полю класу, то через екземпляр класу зробити це вкрай проблематично. Аби прояснити ситуацію, далі ми розглянемо декілька прикладів.

У лістингу 6.7 наведено програмний код, який у певному сенсі дозволяє зрозуміти різницю між полем об'єкта класу і полем екземпляра класу.

Лістинг 6.7. Поле об'єкта класу

```
# Створюємо клас
class MyClass:
    # Поле класу
    name="Клас MyClass"
    # Метод для присвоювання значення
    # полю екземпляра класу
    def set(self,n):
        self.nickname=n
    # Метод для відображення значення
    # поля екземпляра класу
    def get(self):
        print("Значення поля:",self.nickname)
```

Phyton

```
# Конструктор
def __init__(self,n):
    # Полю екземпляра класу
    # присвоюється значення
    self.set(n)
    # Відображується повідомлення
    print("Створено екземпляр класу.")
    # Відображується значення поля екземпляра
    self.get()

# Створюється перший екземпляр класу
green=MyClass("Зелений")
# Звертання до поля класу через екземпляр класу
print("Належність:",green.name)
# Створюється другий екземпляр класу
red=MyClass("Червоний")
# Звертання до поля класу через екземпляр класу
print("Належність:",red.name)
# Полю класу присвоюється значення
MyClass.name="Тут могла бути Ваша реклама!"
# Звертання до поля класу через екземпляр класу
print("Запитує Червоний:",red.name)
# Звертання до поля класу через екземпляр класу
print("Запитує Зелений:",green.name)
```

Виконання цього програмного коду дає такий результат:



Результат виконання програми (з лістингу 6.7)

```
Створено екземпляр класу.
Значення поля: Зелений
Належність: Клас MyClass
Створено екземпляр класу.
Значення поля: Червоний
Належність: Клас MyClass
Запитує Червоний: Тут могла бути Ваша реклама!
Запитує Зелений: Тут могла бути Ваша реклама!
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

Проаналізуємо особливості програмного коду. В тілі класу MyClass, крім оголошення двометодівкою конструктора, екоманданame="Клас MyClass", якою фактично створюється поле name класу MyClass і цьому полю присвоюється текстове значення "Клас MyClass". Метод set() призначений для присвоювання значення полю екземпляра класу nickname. За допомогою методу get() значення поля nickname екземпляра класу відображується у вікні виводу. Також у класі описано конструктор. Під час виконання коду конструктора полю nickname екземпляра класу присвоюється значення аргументу, переданого конструктору, відображується повідомлення про створення екземпляра класу і значення поля nickname. На цьому код класу MyClass, у принципі, вичерпано. Далі в програмі йдуть команди, які створюють екземпляри класу і виконують різні маніпуляції з полем name класу MyClass.

Перший екземпляр (змінна green) класу MyClass створюємо командою green=MyClass("Зелений"). Для цього екземпляра поле nickname, очевидно, отримує значення "Зелений". Під час виклику конструктора у вікні виводу відображуються повідомлення з текстом "Створено екземпляр класу." і "Значення поля: Зелений".

Одразу після створення екземпляра green йде команда print("Належність:",green.name). У ній звертання (інструкція green.name) до поля name об'єкта класу MyClass виконується через екземпляр класу. У результаті з'являється повідомлення з текстом "Належність: Клас MyClass". Таким чином, значенням інструкції green.name є текст "Клас MyClass" — тобто це значення поля name класу MyClass. Якщо б ми замість інструкції green.name використали MyClass.name, отримали б такий самий результат. У цій конкретній ситуації інструкції green.name і MyClass.name еквівалентні з точки зору результату (але це не означає, що так буде завжди).



Причини такої «еквівалентності» не такі очевидні, як може здатися на перший погляд. Все це ми обговоримо пізніше.

Другий екземпляр (змінна red) класу MyClass створюємо командою red=MyClass("Червоний"). Поле nickname екземпляра red отримує значення "Червоний". Під час виклику конструктора

у вікні виводу відображується текст "Створено екземпляр класу." і "Значення поля: Червоний". Якщо звернутися до поля name класу MyClass через екземпляр класу в команді `print("Належність:", red.name)`, то результатом інструкції `red.name` буде текстове значення поля name об'єкта класу MyClass. Про це свідчить повідомлення Належність: Клас MyClass у вікні виводу. Як і в попередньому випадку, звертання до поля класу через екземпляр класу дає такий самий результат, як це було б за використання інструкції `MyClass.name`. Ще один висновок: який би екземпляр класу ми не використовували, якщо ми звертаємося через екземпляр до поля класу, то отримуємо один і той самий результат — значення поля класу.



Читач, знайомий із мовами C++, Java або C#, таку ситуацію знайде дуже схожою на ситуацію з використанням статичних полів.

Щоб перевірити наші підозри, за допомогою команди `MyClass.name="Тут могла бути Ваша реклама!"` присвоюємо нове значення полю name класу MyClass. Потім виконуються команди `print("Запитує Червоний:", red.name)` і `print("Запитує Зелений:", green.name)`, в яких звертання до поля класу здійснюють через екземпляри класу. В результаті виконання цих команд у вікні виводу з'являються відповідно такі повідомлення: Запитує Червоний: Тут могла бути Ваша реклама! і Запитує Зелений: Тут могла бути Ваша реклама!. Висновок простий: обидві інструкції `green.name` і `red.name` насправді повертають значення поля name класу MyClass. Може здатися, що поле класу — це всього лише спільне поле для всіх екземплярів класу. Але тоді виникає запитання: а що буде, якщо ми спробуємо змінити поле класу, присвоївши йому значення не через посилання на об'єкт класу (наприклад, `MyClass.name`), а через посилання на екземпляр класу (наприклад, `green.name` або `red.name`)? Логічно очікувати, що зміни «відчувають» усі екземпляри класу. Але це не так. Розглянемо програмний код у лістингу 6.8.

 **Лістинг 6.8. Поле об'єкта класу і поле екземпляра класу**

```
# Створюємо клас
class MyClass:
    # Поле name класу
    name="Клас MyClass"
    # Метод для присвоювання значення
    # полю nickname екземпляра класу
    def set(self,n):
        self.nickname=n
    # Метод для відображення значення
    # поля nickname екземпляра класу
    def get(self):
        print("Значення поля:",self.nickname)
    # Конструктор
    def __init__(self,n):
        # Присвоюється значення полю
        # nickname екземпляра класу
        self.set(n)
        # Відображується повідомлення
        print("Створено екземпляр класу.")
        # Відображується значення поля
        # nickname екземпляра класу
        self.get()
# Перший екземпляр (змінна green)
green=MyClass("Зелений")
# Перевіряємо значення поля name
# через екземпляр green
print("Належність:",green.name)
# Другий екземпляр (змінна red)
red=MyClass("Червоний")
# Перевіряємо значення поля name
# через екземпляр red
print("Належність:",red.name)
# Змінюємо значення поля name
# через екземпляр green
green.name="Тут був Зелений"
```

Phyton

```
# Перевіряємо значення поля name
# через екземпляр red
print("Запитує Червоний:", red.name)
# Перевіряємо значення поля name
# через екземпляр green
print("Запитує Зелений:", green.name)
# Змінюємо значення поля name
# через об'єкт класу MyClass
MyClass.name="Тут могла бути Ваша реклама!"
# Перевіряємо значення поля name
# через екземпляр red
print("Запитує Червоний:", red.name)
# Перевіряємо значення поля name
# через екземпляр green
print("Запитує Зелений:", green.name)
# Видаляємо поле name екземпляра green
del green.name
# Перевіряємо значення поля name
# через екземпляр green
print("Запитує Зелений:", green.name)
```

Результат виконання цього програмного коду наведено нижче:



Результат виконання програми (з лістингу 6.8)

Створено екземпляр класу.

Значення поля: Зелений

Належність: Клас MyClass

Створено екземпляр класу.

Значення поля: Червоний

Належність: Клас MyClass

Запитує Червоний: Клас MyClass

Запитує Зелений: Тут був Зелений

Запитує Червоний: Тут могла бути Ваша реклама!

Запитує Зелений: Тут був Зелений

Запитує Зелений: Тут могла бути Ваша реклама!

Розділ 6 . Основи об'єктно-орієнтованого програмування

Щодо класу MyClass, то він фактично такий же, як і в попередньому прикладі. Змінилися лише ті команди, «маніпуляції», які ми виконуємо з полем класу name після створення екземплярів green і red. Обговоримо відповідну частину програмного коду.

Поки ми після створення екземплярів red і green перевіряємо значення поля name класу MyClass за допомогою інструкцій виду red.name і green.name, все відбувається цілком очікувано: в обох випадках отримуємо значення поля name класу MyClass. Цю ситуацію ми обговорювали вище, у попередньому прикладі. А далі виконується команда green.name="Тут був Зелений", якою ми наче намагаємося змінити значення поля name класу MyClass, але звертаючись до поля не через об'єкт класу MyClass, а через екземпляр green. Раніше ми за допомогою інструкції green.name читували значення поля name, тому мавмо деякі підстави розраховувати на успіх. Однак під час виконання команди print("Запитує Червоний:",red.name) отримуємо у вікні виводу повідомлення Запитує Червоний: Клас MyClass. Тобто значення поля name, отримане за посиланням через екземпляр red, не змінилося. Хоча під час виконання команди print("Запитує Зелений:",green.name) у вікні виводу отримуємо цілком очікуване повідомлення Запитує Зелений: Тут був Зелений. Поворот подій досить неочікуваний: поки ми через інструкцію green.name значення полю name не присвоювали, значення виразів green.name і red.name збігалися, але після присвоювання значення полю ці інструкції повертають різні значення. Пояснення таке.

Під час виконання команди green.name="Тут був Зелений" насправді значення полю name класу MyClass не присвоюється. Замість цього створюється поле з назвою name в екземпляра green, і створеному полю присвоюється значення "Тут був Зелений". І після цього кожного разу, коли ми будемо використовувати інструкцію green.name, вона буде означати не поле name класу MyClass, а поле name екземпляра green. Іншими словами, поле name екземпляра green «перекриває» поле name класу MyClass. Але це тільки для екземпляра green. У екземпляра red поля name немає, тому інструкція red.name при зчитуванні значення означає поле name класу MyClass.



Насправді, принцип такий. Якщо під час зчитування значення поля через посилання на екземпляр класу виявляється, що таке поле в екземпляра класу `€`, то значення цього поля й повертається. Якщо в екземпляра класу поля з такою назвою немає, то починається пошук однайменного поля серед полів класу.

Під час виконання команди `MyClass.name="Тут могла бути Ваша реклама!"` змінюється значення поля `name` класу `MyClass`. Тому коли в команді `print("Запитує Червоний:", red.name)` використовується посилання `red.name`, то результатом цієї інструкції буде нове значення поля `name` класу `MyClass`. А значення інструкції `green.name` в команді `print("Запитує Зелений:", green.name)` — це значення "тут був Зелений" поля `name` екземпляра `green`.

На наступному етапі командою `del green.name` в екземпляра `green` видаляється поле `name`. Після цього в екземпляра `green` поля `name` вже немає, і значенням інструкції `green.name` в команді `print("Запитує Зелений:", green.name)` є значення поля `name` класу `MyClass`.

Додавання й видалення полів

Віллі, я знаю правило: ми не їмо членів нашої родини.

З телесеріалу «Альф»

Після наших сміливих експериментів з полями об'єкта класу й екземплярів класу читач, найімовірніше, уже здогадався, що поля можна додавати й видаляти, причому це справедливо як для об'єкта класу, так і для екземплярів класу. У цьому пункті ми обговоримо й проаналізуємо дві позиції:

- поля екземпляра класу можна додавати і видаляти після створення екземпляра класу;
- поля об'єкта класу можна додавати і видаляти після створення об'єкта класу.

Із прикладної точки зору, ситуація загалом нескладна: для додавання поля екземпляра класу цьому полю присвоюється значення (розуміло, через посилання на екземпляр класу). Приблизно те ж саме відбувається під час додавання поля об'єкта класу: такому полю присвоюється значення, і цього достатньо, аби в об'єкта класу з'явилося нове поле. Щоб видалити поле в об'єкта класу, після оператора `del` указуємо посилання на поле об'єкта класу (у «крапковому» форматі: клас і через крапку ім'я поля). Аналогічно видаляється й поле екземпляра класу, тільки тепер після оператора `del` указується посилання на поле екземпляра класу (екземпляр класу, крапки й ім'я поля). Невеликий приклад манипуляцій з полями об'єкта класу й полями екземплярів класу наведено в лістингу 6.9.

Лістинг 6.9. Додавання і видалення полів

```
# Створюємо клас
class MyClass:
    pass
# Створюємо екземпляр А
A=MyClass()
# Створюємо екземпляр В
B=MyClass()
# Екземпляру А додаємо
# поле first
A.first="Екземпляр А"
# Екземпляру В додаємо
# поле second
B.second="Екземпляр В"
# Класу MyClass додаємо
# поле total
MyClass.total="Клас MyClass"
# Перевіряємо доступ до полів total
# і first через посилання на екземпляр А
print(A.total,"->",A.first)
# Перевіряємо доступ до поля second
# через посилання на екземпляр А
try:
    # Якщо поле second є
    print(A.second)
# Якщо такого поля немає
except AttributeError:
    print("Такого поля в екземпляра А немає!")
# Перевіряємо доступ до полів total
# і second через посилання на екземпляр В
print(B.total,"->",B.second)
# Перевіряємо доступ до поля first
# через посилання на екземпляр В
try:
    # Якщо поле first є
    print(B.first)
# Якщо такого поля немає
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

```
except AttributeError:  
    print("Такого поля в екземпляра В немає!")  
# Видаляємо поле total класу MyClass  
del MyClass.total  
# Перевіряємо доступ до поля total  
# через посилання на екземпляр А  
try:  
    # Якщо поле є  
    print(A.total)  
# Якщо поля немає  
except AttributeError:  
    print("Такого поля немає!")  
# Перевіряємо доступ до поля total  
# через посилання на екземпляр В  
try:  
    # Якщо поле є  
    print(B.total)  
# Якщо поля немає  
except AttributeError:  
    print("Такого поля немає!")  
# Видаляємо поле first екземпляра А  
del A.first  
# Перевіряємо доступ до поля first  
# через посилання на екземпляр А  
try:  
    # Якщо поле є  
    print(A.first)  
# Якщо поля немає  
except AttributeError:  
    print("Такого поля в екземпляра А немає!")  
.....
```

Результат виконання програмного коду такий:



Результат виконання програми (з листингу 6.9)

```
Клас MyClass -> Екземпляр А  
Такого поля в екземпляра А немає!  
Клас MyClass -> Екземпляр В
```

Phyton

Такого поля в екземпляра В немає!

Такого поля немає!

Такого поля немає!

Такого поля в екземпляра А немає!

У цьому прикладі створено клас MyClass, який не містить, власне, нічого. На основі цього класу створюють два екземпляри з назвами А і В. Потім командами і A.first="Екземпляр А", B.second="Екземпляр В", і MyClass.total="Клас MyClass" додаємо (із присвоюванням значень) поле first для екземпляра А, поле second для екземпляра В і поле total для класу MyClass. У команді print(A.total,"->",A.first) є посилання на поле first екземпляра класу і поле total об'єкта класу, причому обидва посилання виконуються через екземпляр А. За результатом виконання команди (повідомлення Клас MyClass -> Екземпляр А у вікні виводу) бачимо, що проблем із доступом до полів не виникає. У принципі, дещо схоже ми спостерігали в попередніх прикладах, але тут є одна особливість. Пов'язана вона з тим, що поле total з'явилось у класу MyClass уже після того, як були створені екземпляри А і В. Втім, доступ до доданого поля total через екземпляри класу є. А ось додавання поля одному екземпляру класу для другого екземпляра «залишається непоміченим» (що, у принципі, цілком логічно). Під час спроби виконати команду print(A.second) виникає помилка класу AttributeError: поля second в екземпляра А немає. Цю ситуацію ми обробляємо за допомогою блоку try-except. Результатом є повідомлення Такого поля в екземпляра А немає!, яке відображується командою print("Такого поля в екземпляра А немає!") в except-блоці.

Аналогічні справи з екземпляром В: доступ до поля total класу MyClass і поля second екземпляра через змінну В отримуємо без проблем, а до поля first екземпляра доступу немає, оскільки це поле додавалося в екземпляр А, і екземпляр В до цього поля жодного відношення не має.

На наступному етапі за допомогою команди del MyClass.total видаляємо поле total об'єкта класу MyClass, після чого за допомогою команди print(A.total) (у try-блоці) намагаємося прочитати значення цього поля. Судячи з того, що в цьому випадку виникає помилка класу AttributeError (неправильний атрибут) і в except-блоці виконується команда print("Такого поля немає!"), поле total дійсно видалено

Розділ 6 . Основи об'єктно-орієнтованого програмування

з об'єкта класу MyClass. Немає доступу до видаленого поля total і через змінну B.

Нарешті, за допомогою команди del A.first видаляємо поле first екземпляра A. Як наслідок, під час спроби виконати команду print(A.first) у try-блоці виникає помилка. Тому в except-блоці виконується команда print("Такого поля в екземпляра A немає!").



Щоб легше було зрозуміти, що і як відбувається під час додавання й видалення полів об'єкта класу й екземплярів класу, слід урахувати, що поля та їхні значення для кожного екземпляра класу й об'єкта класу зберігаються в словнику. Поле є ключем, а значення поля — елементом словника. Під час звертання до поля екземпляра пошук спочатку виконується за словником екземпляра класу, і якщо там немає відповідного ключа (нагадаємо, що це насправді назва поля), то пошук зміщується до словника з назвами і значеннями полів об'єкта класу. Додаючи або видаляючи поля, ми вносимо зміни до відповідного словника. Тому екземпляри, створені до внесення змін в об'єкт класу, «знають» про ці зміни.

Методи і функції

Пожежна? Скоріше приїжджаєте: на мене напав велетенський тарган!

З телесеріалу «Альф»

У цьому розділі ми обговоримо методи. Тема ця нетривіальна, але ми постараемся виокремити основні, так би мовити, концептуальні моменти, які дозволяють читачеві більш-менш вільно орієнтуватися в тому багатстві прийомів і підходів, які є в його розпорядженні при програмуванні на Python. Почнемо з невеликого прикладу, наведеного в лістингу 6.10.



Лістинг 6.10. Метод екземпляра і функція класу

```
# Створюємо клас
class MyClass:
    # Метод екземпляра
    def say(self):
        # Відображується повідомлення
        print("Усім вітання!")

# Створюємо екземпляр класу
obj=MyClass()
# Викликаємо метод екземпляра.
# Аргументів немає
obj.say()
# Викликаємо функцію класу.
# Аргумент - екземпляр класу
MyClass.say(obj)
# Викликаємо функцію класу.
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

```
# Аргумент - текст  
MyClass.say("Якийсь текст")
```

Результат виконання програмного коду наведено нижче:

■ Результат виконання програми (з лістингу 6.10)

```
Усім вітання!  
Усім вітання!  
Усім вітання!
```

Ми створюємо клас MyClass, у якому описуємо метод екземпляра класу say(). У тілі методу лише одна команда print("Усім вітання!"), яка, очевидно, у вікні виводу відображує повідомлення. Хоча в методу є аргумент self, цей аргумент явно не використовується.

У цьому класі немає нічого незвичайного. Однак ми все ж таки проведемо невеликий експеримент. Для цього за допомогою команди obj=MyClass() створюємо екземпляр obj. Потім із екземпляра викликаємо метод say() (мається на увазі команда obj.say()). Результат цілком очікуваний: відображується повідомлення Усім вітання!. Потім послідовно виконуються команди MyClass.say(obj) і MyClass.say("Якийсь текст"). Результат такий самий, як під час виконання команди obj.say(). Спробуємо розібратися, чому ж відбувається саме так.

Почнемо з команди MyClass.say(obj). У певному сенсі вона є втіленням команди obj.say(). Річ у тім, що коли ми описуємо в класі MyClass метод екземпляра say(), насправді ми описуємо *функцію* say(). Це, взагалі, звичайнісінька функція, просто описується в тілі класу. Раніше ми такі функції називали *методами* і викликали через екземпляр класу. Тут ми викликаємо функцію, вказавши замість екземпляра класу сам клас. Так теж можна робити.



Щоб не запутатися, будемо називати say() функцією, якщо йдеться про команду MyClass.say(obj), і методом, якщо йдеться про команду obj.say().

Із формальної точки зору, головна особливість функції `say()` у тому, що під час звертання до неї ми вказуємо ще й ім'я класу `MyClass`. Виклик функції `say()` через ім'я класу — це як начеб ми викликали звичайну функцію: за допомогою команди `MyClass.say(obj)` викликається функція `say()`, описана в тілі класу `MyClass`, а як аргумент цій функції передається посилання на екземпляр класу `obj`.

Коли ми говоримо про виклик *methodу* `say()`, то маємо на увазі, що виклик здійснюється через посилання на екземпляр класу. Як ми вже знаємо, екземпляр класу неявно передається як перший (і в цьому випадку єдиний) аргумент у `say()`. Тому команда `obj.say()` еквівалентна виклику функції `say()` із класу `MyClass` із аргументом `obj`. Отже, нема нічого дивного в тому, що виконання команд `MyClass.say(obj)` і `obj.say()` має однакові наслідки. Щодо команди `MyClass.say("Якийсь текст")`, то в певному сенсі це «хуліганство», хоча й цілком законне. Ми викликаємо функцію `say()` із класу `MyClass` із аргументом, який є текстовим значенням, хоча, за логікою, аргументом повинне би бути посилання на екземпляр класу. Але оскільки в тілі функції `say()` посилання на екземпляр класу явно не використовується (тобто аргумент у функції є, але в тілі функції не використовується), то немає особливого значення, що ж ми передаємо аргументом функції — головне, щоб аргумент просто був.

Узагалі, ситуація не така проста, як може здатися на перший погляд. Щоб її трохи прояснити, є сенс нагадати, що функція в Python — це деякий об'єкт (у загальному значенні цього терміна), а якщо точніше, то об'єкт типу `function`. Тому функцію, описану в класі, можна розглядати саме як такий об'єкт — за аналогією до того, як ми описували в тілі класу змінні, які відігравали роль полів об'єкта класу. Ім'я функції, описаної в тілі класу, схоже на ім'я звичайної змінної, описаної в тілі класу. Як і у випадку з полем об'єкта класу, при посиланні на функцію, описану в тілі класу, ім'я класу через крапку вказується перед іменем функції. У контексті сказаного інструкція `MyClass.say` є посиланням на об'єкт типу `function`.



Хто бажає, може провести такий експеримент: у програмний код, описаний вище, додати команду `print(type(MyClass.say))`. У результаті

Розділ 6 . Основи об'єктно-орієнтованого програмування

у вікні виводу з'явиться повідомлення <class 'function'>. Результатом виконання команди `print(type(obj.say))` буде повідомлення <class 'method'>.

Ми можемо розглядати ім'я функції як деякий атрибут класу, до якого звичай звертаються з використанням круглих дужок (і аргументів у них). Зокрема, якщо функція, описана в тілі класу, викликається через посилання на клас (тобто у форматі `клас.функція(аргументи)`), то фактично мається на увазі виклик функції з відповідними аргументами. За великим рахунком, тут ідеться про звичайну функцію, але тільки «прив'язану» до класу. У такому контексті ми можемо описати в тілі класу практично будь-яку функцію й викликати її як звичайну функцію, але тільки використовуючи в назві явне посилання на клас. Із іншого боку, зрозуміло, що звичай функції в класі описують не для цього. При наймні раніше ми викликали функції, описані в тілі класу, через посилання на екземпляр класу й називали такі функції *методами*. Тут, зрозуміло, нема протиріччя. Функції, описані в тілі класу, можна викликати не тільки через клас, але й через екземпляр класу. У певному сенсі, при цьому ми можемо говорити про виклик тієї самої функції, що й під час виклику через ім'я класу, але через екземпляр класу звертання до функції здійснюється не так «прямолінійно».

Що ж відбувається (у загальних рисах), коли ми викликаємо функцію через екземпляр класу (тобто викликаємо метод)? Іншими словами, як функція стає методом? А відбувається приблизно таке.

- Серед назв функцій, описаних у тілі класу, виконується пошук імені методу, який викликається з екземпляра класу. Припускаємо, що функцію з відповідною назвою знайдено (якщо не знайдено, то виникає помилка).
- Створюється об'єкт методу (об'єкт класу `method`). Цей об'єкт відповідає функції, що викликається з аргументами, які передаються методу під час виклику, але додатково як перший аргумент додається посилання на екземпляр класу, з якого викликається метод.



Простіше кажучи, команда вигляду `екземпляр.ім'я(аргументи)` еквівалентна команді `клас.ім'я(екземпляр, аргументи)`.

Далі для нас найважливішими будуть дві позиції:

- під час виклику методу першим аргументом неявно передається посилання на екземпляр класу;
- ім'я функції/методу є атрибутом, якому можна присвоїти значення.

Щодо першої позиції, то вона нам була відома і до цього. Стосовно другої позиції потрібні деякі пояснення. Представити їх краще за все на конкретному прикладі, що ми й зробимо. Наприклад, зовсім необов'язково описувати метод екземпляра класу або функцію класу безпосередньо в тілі класу. Ми можемо описати відповідну функцію окремо від опису класу, а потім «зареєструвати» її як метод екземпляра або як функцію класу. За допомогою інструкції `def` метод екземпляра класу або функцію класу можна видалити. Щоб зrozуміти, як виконуються такі операції, розгляньмо лістинг 6.11.

Листинг 6.11. Додавання й видалення методів

```
# Створюємо клас
class MyClass:
    pass

# Створюємо екземпляри класу
A=MyClass()
B=MyClass()
C=MyClass()

# Створюємо першу функцію
def hello():
    print("Метод екземпляра - 'hello'")

# Створюємо другу функцію
def hi():
    print("Ще один метод - 'hi'")

# Визначаємо метод екземпляра
A.say=hello
# Визначаємо метод екземпляра
C.say=hi
# Викликаємо метод екземпляра
A.say()
# Викликаємо метод екземпляра
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

```
try:  
    B.say()  
    # Якщо такого методу немає  
except AttributeError:  
    print("Такого методу немає")  
    # Викликаємо метод екземпляра  
C.say()  
    # Викликаємо функцію класу  
try:  
    MyClass.say()  
    # Якщо такої функції немає  
except AttributeError:  
    print("Такої функції немає")  
    # Видаляємо метод екземпляра  
del A.say  
    # Викликаємо метод екземпляра  
try:  
    A.say()  
    # Якщо такого методу немає  
except AttributeError:  
    print("Такого методу немає")  
    # Викликаємо метод екземпляра  
C.say()
```

Результат виконання цього програмного коду такий:



Результат виконання програми (з листингу 6.11)

```
Метод екземпляра - 'hello'  
Такого методу немає  
Ще один метод - 'hi'  
Такої функції немає  
Такого методу немає  
Ще один метод - 'hi'
```

Ми створюємо клас із традиційною назвою `MyClass`. У класі в цьому конкретному випадку нічого не описано (ні полів, ані методів). Далі на основі класу створюються три екземпляри: `A`, `B` і `C`. Також ми описуємо дві

функції `hello()` і `hi()`, в яких немає аргументів і які не повертають результат. Під час виконання кожної з цих функцій в області виводу відображується повідомлення (дляожної функції свое). До цього моменту не відбувається нічого незвичайного. Відверта «екзотика» починається з команди `A.say=hello`, якою атрибуту `say` екземпляра `A` присвоюється значенням ідентифікатор `hello` — ім'я однієї зі створених нами функцій. Що все це означає і що при цьому відбувається? Для відповіді на ці запитання варто згадати, що функція реалізується через спеціальний об'єкт, а ім'я функції — це посилання на цей об'єкт. Тому в цьому разі ідентифікатор `hello` є посиланням на об'єкт, через який реалізується функція `hello()`. З іншого боку, якщо атрибуту `say` екземпляра `A` присвоюється посилання на об'єкт, що реалізує функцію, то зазначений атрибут буде посилюватися на цей об'єкт. Отже, атрибут `say` можна розглядати як функцію, причому це та ж функція, що й функція `hello()`. Тому, коли виконується команда `A.say()`, насправді викликається функція `hello()`.

Дещо схоже відбувається під час виконання команди `C.say=hi`, із по правкою лише на те, що йдеться про екземпляр `C` (а не `A`) і функцію `hi()` (а не `hello()`). У результаті під час виконання команди `C.say()` викликається функція `hi()`. А ось під час виконання команди `B.say()` виникає помилка (типу `AttributeError`), оскільки атрибут `say` в екземпляра `B` немає. Достоту так, як немає такого атрибута `say` в об'єкта класу `MyClass`. Тому команда `MyClass.say()` теж викликає помилку.

Щоб видалити атрибут `say` в екземпляра `A`, використовуємо команду `del A.say`. Після цього команда `A.say()` викликатиме помилку, а команда `C.say()` — ні, оскільки в екземпляра `C` атрибут `say` не видалявся.



Стверджувати, що ми, присвоюючи атрибутам екземплярів класу значенням посилання на функції, створювали тим самим методи екземплярів, буде не зовсім коректно. Є декілька важливих обставин, які суттєво охолоджують наш запал. По-перше, функції, посилання на які ми присвоювали атрибутам, не мають доступу до екземплярів класу. Іншими словами, ми не зможемо змінити програмний код функції `hello()` так, щоб під час виконання команди `A.say()` був прямий доступ до екземпляра `A`. Як би не хотілося нам розглядати `say()` як метод екземпляра класу, цей метод фактично до екземпляра класу доступу не має. Це серйозний мінус. Далі, якщо б ми взяли декілька екземплярів і атрибуту `say` кожного з цих екземплярів присвоїли посилання

Розділ 6 . Основи об'єктно-орієнтованого програмування

на функцію `hello()`, то всі ці екземпляри через атрибут `say` посилалися б на одну й ту ж функцію. Виходить «загальна» функція для всіх екземплярів. У тому, що це дійсно функція, а не метод, легко переконатися — достатньо після команди `A.say=hello` додати команду `print(type(A.say))`. У вікні виводу з'явиться повідомлення `<class 'function'>`. Нагадаємо, що коли перевіряється тип методу екземпляра, в аналогічній ситуації з'являється повідомлення `<class 'method'>`.

Коротше кажучи, тим, що ми робили вище, краще все ж особливо не захоплюватися.

Ефективнішим може бути присвоювання посилання на функцію значенням атрибуту класу (зрозуміло, зворотна процедура — видалення атрибута з посиланням на функцію, — теж допускається). Невеликий приклад наведено в лістингу 6.12.

Лістинг 6.12. Додавання і видалення функції класу

```
# Створюємо клас
class MyClass:
    def __init__(self,n):
        self.name=n
# Створюємо екземпляри класу
A=MyClass("A")
B=MyClass("B")
# Створюємо функцію з аргументом
def hello(self):
    print("Це екземпляр",self.name,"- hello")
# Створюємо функцію з аргументом
def hi(self):
    print(self.name+": hi")
# Визначаємо функцію класу
MyClass.say=hello
# Викликаємо метод екземпляра
A.say()
# Викликаємо метод екземпляра
B.say()
# Викликаємо функцію класу
MyClass.say(A)
```

```
Phyton .....  
  
MyClass.say(B)  
# Змінюємо посилання на функцію  
MyClass.say=hi  
# Викликаємо метод екземпляра  
A.say()  
# Викликаємо метод екземпляра  
B.say()  
# Викликаємо функцію класу  
MyClass.say(A)  
MyClass.say(B)  
# Видаляємо функцію класу  
del MyClass.say  
# Викликаємо метод екземпляра  
try:  
    A.say()  
# Якщо методу немає  
except AttributeError:  
    print("Такого методу немає")  
# Викликаємо метод екземпляра  
try:  
    B.say()  
# Якщо методу немає  
except AttributeError:  
    print("Такого методу немає")  
# Викликаємо функцію класу  
try:  
    MyClass.say(A)  
# Якщо функції немає  
except AttributeError:  
    print("Такої функції немає")  
.....
```

Результат виконання цього програмного коду наведено нижче:



Результат виконання програми (з лістингу 6.12)

```
Це екземпляр А - hello  
Це екземпляр В - hello  
Це екземпляр А - hello
```

Це екземпляр В - hello

A: hi

B: hi

A: hi

B: hi

Такого методу немає

Такого методу немає

Такої функції немає

У класі MyClass описано конструктор. При створенні екземпляра класу конструктору передається аргумент, який визначає значення поля name екземпляра класу. Ми створюємо два екземпляри: A (зі значенням "A" для поля name) і B (зі значенням "B" для поля name). Також описуємо дві функції hello() і hi(). Причому в кожній із функцій оголошено по одному аргументу (аргумент називається self), і неявно припускається, що це посилання на екземпляр класу MyClass, оскільки в тілі функцій є посилання self.name на поле name екземпляра self. Разом із тим, обидві ці функції описано поза тілом класу MyClass, тож говорити про функції класу або метод екземпляра ще рано — рано, поки не виконано команду MyClass.say=hello. У цьому разі атрибуту say класу MyClass присвоюється значенням посилання на функцію hello(). Далі можемо викликати метод say() для екземплярів класу командами A.say() і B.say(). При цьому фактично викликається функція hello(), аргументом якій передається посилання на екземпляр класу, з якого викликається метод say(). Analogічні результати отримуємо, виконуючи команди MyClass.say(A) і MyClass.say(B). У них безпосередньо викликається функція say() класу MyClass із одним аргументом.

Ми можемо змінити значення атрибута say класу MyClass, виконавши, наприклад, команду MyClass.say=hi. Після виконання цієї команди під час виклику функції класу або методу екземпляра say() реально буде виконуватися код функції hi(). В останньому легко пerekнатися за допомогою команд A.say(), B.say(), MyClass.say(A) і MyClass.say(B). Зрештою, ми можемо видалити функцію say() класу MyClass за допомогою команди del MyClass.say. Після цього виконання перерахованих вище команд із викликом функції/методу say() призводить до помилки.

Можливі й інші варіанти «дій» із функціями класу й методами екземпля-
рів класу. Деякі невеликі приклади наведено в лістингу 6.13.

Лістинг 6.13. Операції з методами і функціями

```
# Створюємо клас
class MyClass:
    # Конструктор екземпляра класу
    def __init__(self,n):
        self.name=n
    # Метод екземпляра класу
    def say(self):
        print("Клас MyClass:",self.name)
# Створюємо екземпляри класу
A=MyClass ("A")
B=MyClass ("B")
# Викликаємо метод екземпляра
A.say()
B.say()
# Посилання на метод записуємо в змінну
F=A.say
# Викликаємо функцію
F()
# Атрибуту екземпляра присвоюється текст
A.say="Поле екземпляра A"
# Перевіряємо значення поля
print(A.say)
# Намагаємося викликати метод екземпляра
try:
    A.say()
# Якщо такого методу немає
except TypeError:
    print("Неправильна команда")
# Викликаємо метод екземпляра
B.say()
# Викликаємо функцію
F()
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

У класі MyClass описано конструктор екземпляра класу, в якому полю name екземпляра присвоюється значення. Також у класі описано метод say(), що відображає текст і значення поля name екземпляра, з якого викликається метод. Під час створення екземпляри A і B отримують для своїх полів name відповідно значення "A" і "B". У кожного з екземплярів A і B є метод say(), у чому нескладно переконатися за допомогою команд A.say() і B.say().

Далі міркуватимемо так. Інструкція A.say є посиланням на об'єкт методу say() екземпляра A класу MyClass. Ніхто не забороняє нам присвоїти посилання на цей об'єкт у деяку змінну. Саме так ми і робимо, коли використовуємо команду F=A.say. У результаті її виконання в змінну F записано посилання на об'єкт — той самий, на який посилається інструкція A.say. Оскільки йдеться про об'єкт методу, то викликати цей метод можемо за допомогою інструкції A.say() або F(). В обох випадках викликається один і той же метод, тому ми отримуємо один і той же результат.

На наступному кроці командою A.say="Поле екземпляра A" атрибуту say екземпляра A присвоюється текстове значення. Хоча раніше атрибут say екземпляра A посылався на метод екземпляра, тепер це фактично поле, яке посилається на текст. Тому викликати метод екземпляра за допомогою команди A.say() більше не вийде. Якщо ми скористаємося командою A.say(), виникне помилка неузгодженості типів TypeError. Коректним є звертання A.say до текстового поля екземпляра.



Таким чином, в екземпляра A з'являється поле say, і це поле «перекриває» метод say() екземпляра. Зрозуміло, що це загальне правило.

При цьому в екземпляра B метод say() «залишається» і без проблем може бути викликаний командою B.say(). Але найцікавіше, що за допомогою інструкції F() ми, як і раніше, можемо викликати метод say() екземпляра A. Пояснення просте: змінна F посилається на об'єкт методу say() екземпляра A. Раніше на цей об'єкт містилося посилання і в атрибуті A.say. І хоча після присвоювання текстового значення атрибуту say інструкцію A.say() для виклику методу не можна використовувати, змінна F для цієї мети цілком підійде. У результаті після виконання всього програмного коду отримуємо такий результат:

■ Результат виконання програми (з лістингу 6.13)

```
Клас MyClass: А
Клас MyClass: В
Клас MyClass: А
Поле екземпляра А
Неправильна команда
Клас MyClass: В
Клас MyClass: А
```

Хоча описані вище можливості з маніпулювання полями й методами досить унікальні (такі «трюки» можна робити далеко не в кожній мові програмування), картина була б неповною без одного дуже важливого механізму, без якого взагалі не було б сенсу розглядати ООП. Йдеться про **наслідування**. Наслідування, як і деякі інші моменти (наприклад, перевантаження операторів), розглядається в наступному розділі.

Копіювання екземплярів і конструктор створення копії

Якщо я буду потрібен — я біля
холодильника.

З телесеріалу «Альф»

Конструктори ми вже обговорювали і знаємо, що конструктору екземпляра класу можуть передаватися аргументи. Це з одного боку. З іншого боку, є у певному сенсі «класична» задача, яка полягає в тому, що на основі одного екземпляра класу треба створити дослідно такий же. Грубо кажучи, актуальною є задача створення копії екземпляра класу. Проблема в тому, що звичайним присвоюванням значення змінним, які посилаються на екземпляри класів, тут не обійтися. Адже під час присвоювання значення змінним просто «перекидаються» посилання. Так, якщо якось змінна `x` посилається на екземпляр класу, то після виконання команди `y=x` змінна `u` буде посилатися на той самий екземпляр, що й змінна `x`. І це не те, що нам треба. Нам треба, образно кажучи, щоб змінна `u` посилається на екземпляр з такими самими характеристиками, як в екземпляра, на який посилається змінна `x`.



Тобто нам начебто потрібен достоту такий же екземпляр, але інший.

У принципі, для створення копії екземпляра можна скористатися функціями `copy()` або `deepcopy()` із модуля `copy`. При використанні функції `copy()` створюється *поверхнева копія* екземпляра, у той час як функція `deepcopy()` дозволяє створювати *повні копії*. Відмінність між поверхневою та повною копіями проявляється, коли серед полів екземплярів

є змінні, що посилаються на дані *змінюваних типів* (прикладом можуть бути списки). Невелику ілюстрацію до використання цих функцій подано в лістингу 6.14.

Лістинг 6.14. Створення копії екземпляра

```
# Імпорт функцій copy() і deepcopy()
# з модуля copy
from copy import copy, deepcopy
# Клас
class MyClass:
    # Конструктор
    def __init__(self, name, nums):
        # Значення поля name
        self.name=name
        # Значення поля nums
        self.nums=nums
    # Метод для відображення
    # значень полів екземпляра
    def show(self):
        # Поле name
        print("name ->",self.name)
        # Поле nums
        print("nums ->",self.nums)
# Створення екземпляра класу
x=MyClass("Python", [1,2,3])
print("Екземпляр x:")
# Відображення полів екземпляра x
x.show()
# Поверхнева копія екземпляра x
y=copy(x)
# Повна копія екземпляра x
z=deepcopy(x)
print("Екземпляр y:")
# Відображення полів екземпляра y
y.show()
print("Екземпляр z:")
# Відображення полів екземпляра z
```

```
z.show()
print("Поля екземпляра x змінюються!")
# Зміна значення поля name
# екземпляра x
x.name="Java"
# Зміна значення елемента в списку
# nums - поле екземпляра x
x.nums[0]=0
print("Екземпляр x:")
# Відображення полів екземпляра x
x.show()
print("Екземпляр y:")
# Відображення полів екземпляра y
y.show()
print("Екземпляр z:")
# Відображення полів екземпляра z
z.show()
```

У результаті виконання програмного коду отримуємо таке:

 **Результат виконання програми (з листингу 6.14)**

```
Екземпляр x:
name -> Python
nums -> [1, 2, 3]
Екземпляр y:
name -> Python
nums -> [1, 2, 3]
Екземпляр z:
name -> Python
nums -> [1, 2, 3]
Поля екземпляра x змінюються!
Екземпляр x:
name -> Java
nums -> [0, 2, 3]
Екземпляр y:
name -> Python
nums -> [0, 2, 3]
```

Phyton

Екземпляр z:
name -> Python
nums -> [1, 2, 3]

Імпорт функцій `copy()` і `deepcopy()` з модуля `copy` виконується за допомогою інструкції `from copy import copy, deepcopy`. Клас, над екземплярами якого будуть проводитися експерименти з «клонування», називається `MyClass`. У конструкторі екземпляра класу визначаються поля `name` і `nums`. Присвоєні полям значення передаються як аргументи конструктору. Ми припускаємо, що поле `name` є текстовим, а поле `nums` є числовим списком (хоча це, звичайно, умовно).

У класі описано метод екземпляра `show()`, за допомогою якого виконується відображення полів `name` і `nums` екземпляра класу. На цьому опис класу закінчується. Далі переходимо до процедури створення копій екземплярів. Для цієї мети нам потрібен вихідний екземпляр класу, який буде копіюватися. Екземпляр класу створюємо за допомогою команди `x=MyClass("Python", [1,2,3])`. У результаті створюється екземпляр `x` класу `MyClass`. У створеного екземпляра класу значенням поля `name` є текст `"Python"`, а значенням поля `nums` — список `[1,2,3]`. Перевірити значення полів екземпляра `x` можна за допомогою команди `x.show()`.

Копії екземпляра `x` створюються так: команда `y=copy(x)` створює поверхневу копію (екземпляр `y`) екземпляра `x`, а команда `z=deepcopy(x)` створює повну копію (екземпляр `z`) екземпляра `x`. На цьому етапі значення полів у екземплярів `y` і `z` такі ж, як в екземпляра `x`: підтвердженням є результат виконання команд `y.show()` і `z.show()`.

На наступному етапі командами `x.name="Java"` і `x.nums[0]=0` змінюються значення поля `name` екземпляра `x` і перший (із нульовим індексом) елемент у списку `nums`, що є полем цього ж екземпляра. За допомогою команд `x.show()`, `y.show()` і `z.show()` перевіряємо, чи змінилися (і якщо так, то як саме) поля екземплярів `x`, `y` і `z`. Із екземпляром `x` все просто. Його поля змінилися строго відповідно до тих команд, які були виконані під час присвоювання значень. Поля екземпляра `z` (повна копія) не змінилися. Щодо екземпляра `y`, то у нього не змінилося поле `name`, але змінилося поле-список `nums` (перший елемент став нульовим, як і в екземпляра `x`). Пояснення в тому, що при створенні поверхневої

копії для змінюваних типів, таких як списки, виконується копіювання посилань, але не значень. Тому реально екземпляри `x` і `y` (поверхнева копія `x`) посилаються через свої поля `nums` на один і той же список, тоді як в екземпляра `z` (повна копія `x`) поле `nums` «персональне».

Хоча описаний вище підхід легітимний, але все ж не завжди прийнятний (унаслідок різних причин). Існують й інші варіанти. Зокрема, ми можемо подивитися на ситуацію ширше й переформулювати задачу про створення копії екземпляра в задачу про створення екземпляра класу на основі вже існуючого екземпляра. Якщо при цьому вимагати, щоб створений екземпляр мав такі ж значення полів вихідного екземпляра, то йтиметься про створення копії. З практичної точки зору це означає, що в класі необхідно описати конструктор екземпляра, аргументом якому передається посилання на інший екземпляр того ж класу. За традицією, означений конструктор називається **конструктором створення копії** (хоча при цьому може створюватися зовсім не копія).



Проблема в тому, що в Python немає перевантаження методів. Тому ми не можемо, як у C++ або Java, створити декілька конструкторів різних видів. У Python в екземпляра класу конструктор повинен бути один. Це вносить певну інтригу в процес створення конструктора копії.

На практиці дуже зручно, якщо екземпляри класу можуть створюватися різними способами. Насамперед, мається на увазі можливість передавати конструктору під час створення екземпляра класу різні набори аргументів. Це ж зауваження, до речі, стосується й звичайних методів: добре, коли метод може викликатися з різними аргументами. У таких мовах програмування, як C++, Java і C#, проблема знімається за допомогою механізму *перевантаження методів і функцій*. У мові Python як такого механізму перевантаження методів немає. Але, відверто кажучи, враховуючи виключну «гнучкість» мови Python, особливої необхідності в перевантаженні не спостерігається. Існують інші можливості. Одна з них — створення функцій і методів зі змінною кількістю аргументів. Цей підхід описується в останньому розділі. Тут ми розглянемо приклад класу з конструктором, якому можна передавати різну кількість аргументів, включаючи й випадок, коли як аргумент передається посилання на вже існуючий екземпляр класу. Механізм перевантаження нам удасться «обійти» завдяки призначенню аргументам значень за замовчуванням

і скориставшись перевіркою типу переданого конструктору аргументу. Відповідний програмний код наведено в лістингу 6.15.

Лістинг 6.15. Конструктор створення копії екземпляра

```
# Клас
class ComplNum:
    # Конструктор створення екземпляра класу
    def __init__(self,x=0,y=0):
        # Якщо аргумент x - екземпляр
        # класу ComplNum
        if type(x)==ComplNum:
            # Значення поля Re
            self.Re=x.Re
            # Значення поля Im
            self.Im=x.Im
        # Якщо аргумент x - не екземпляр
        # класу ComplNum
        else:
            # Значення поля Re
            self.Re=x
            # Значення поля Im
            self.Im=y
    # Метод для відображення значень
    # полів екземпляра класу
    def show(self):
        print("Re =",self.Re)
        print("Im =",self.Im)
# Створюється екземпляр класу
a=ComplNum(1,2)
# Створюється копія екземпляра класу
b=ComplNum(a)
print("Екземпляр a:")
# Значення полів вихідного екземпляра
a.show()
print("Екземпляр b:")
# Значення полів екземпляра-копії
b.show()
```

Розділ 6 . Основи об'єктно-орієнтованого програмування

```
print("Поля екземпляра а змінюються!")
# Змінюємо значення полів вихідного
# екземпляра
a.Re=10
a.Im=20
print("Екземпляр а:")
# Значення полів вихідного екземпляра
a.show()
print("Екземпляр b:")
# Значення полів екземпляра-копії
b.show()
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 6.15)

```
Екземпляр а:
Re = 1
Im = 2
Екземпляр b:
Re = 1
Im = 2
Поля екземпляра а змінюються!
Екземпляр а:
Re = 10
Im = 20
Екземпляр b:
Re = 1
Im = 2
```

У наведеному прикладі ми створюємо клас `ComplNum` зі слабким натяком на реалізацію комплексних чисел за допомогою екземплярів цього класу. У всікому випадку, в екземплярів класу є два поля: `Re` і `Im` (як наочній й уявна частини комплексного числа). При створенні екземпляра класу аргументами конструктору можна передати значення для полів `Re` і `Im`, а можна передати посилання на екземпляр класу, на основі якого буде створюватися копія.



Конструктор описано з трьома аргументами. Перший аргумент `self` є посиланням на екземпляр класу (той екземпляр, що створюється). Ще два аргументи позначені як `x` і `y`. У кожного з цих аргументів є нульові значення за замовчуванням. Тому формально конструктор можна викликати без аргументів, із одним аргументом або з двома аргументами. Якщо конструктор викликається без аргументів, то це все одно, що він викликався б із двома нульовими аргументами. Якщо конструктору передано тільки один аргумент, то другий уважається нульовим.

У тілі конструктора в умовному операторі перевіряється тип першого аргументу. Точніше, перевіряється умова `type(x) == ComplNum`, яка полягає в тому, що тип даних, на які посилається аргумент `x`, є екземпляром класу `ComplNum`. Якщо це так, то аргумент `x` обробляється як екземпляр класу `ComplNum`. Командами `self.Re=x.Re` і `self.Im=x.Im` значення полів `Re` і `Im` екземпляра `x` присвоюються відповідно полям `Re` і `Im` екземпляра `self` (екземпляр, який створюється під час виклику конструктора).

Якщо умова `type(x) == ComplNum` хибна, то ми неявно припускаємо, що `x` і `y` — числові аргументи. У цьому разі командами `self.Re=x` і `self.Im=y` значення аргументів присвоюються полям `Re` і `Im` створюваного екземпляра класу.



Отже, якщо першим (але не обов'язково єдиним аргументом) конструктору передано посилання на екземпляр класу `ComplNum`, то другий аргумент конструктора в обчисленнях не використовується — незалежно від того, переданий він явно чи ні.

Також у класі `ComplNum` описано метод екземпляра `show()`, який відображає в області виводу значення полів екземпляра. Ми цей метод використовуємо для перевірки «вмісту» екземплярів класу `ComplNum`.

Поза кодом класу `ComplNum` командою `a=ComplNum(1, 2)` створюємо екземпляр `a` класу `ComplNum`. Копію цього екземпляра створюємо командою `b=ComplNum(a)`. Тут як аргумент конструктору екземпляра класу `ComplNum` передано посилання на раніше створений екземпляр `a`. Безпосередньою перевіркою переконуємося, що екземпляр `b` справді є копією

Розділ 6 . Основи об'єктно-орієнтованого програмування

екземпляра *a*: після створення екземпляра *b* у нього такі ж значення полів, як і в екземпляра *a*, а після зміни значень полів екземпляра *a* значення полів екземпляра *b* не змінюються.

Варто зауважити, що навіть під час опису в класі конструктора створення копії, необхідно пам'ятати про особливості копіювання даних змінюваних типів — таких, як списки. Так, програмний код, розглянутий нами під час ілюстрації принципів використання функцій *copy()* і *deepcopy()* з модуля *copy*, можна було б реалізувати дещо інакше, застосувавши конструктор створення копії. Розглянемо листинг 6.16, де показано, як можна описати конструктор, що дозволяє створювати копії екземпляра класу з полями-списками.

Листинг 6.16. Конструктор створення копії й поля-списки

```
# Клас
class MyClass:
    # Конструктор
    def __init__(self,arg,nums=None):
        # Якщо аргумент arg - посилання на
        # екземпляр класу MyClass
        if type(arg)==MyClass:
            # Значення поля name
            self.name=arg.name[:]
            # Значення поля nums
            self.nums=arg.nums[:]
        # Якщо аргумент arg - не посилання на
        # екземпляр класу MyClass
        else:
            # Значення поля name
            self.name=arg
            # Значення поля nums
            self.nums=nums
    # Метод для відображення
    # значень полів екземпляра
    def show(self):
        # Поле name
```

Phyton

```
        print("name ->", self.name)
        # Поле nums
        print("nums ->", self.nums)
# Створення екземпляра класу
x=MyClass("Python", [1,2,3])
print("Екземпляр x:")
# Відображення полів екземпляра x
x.show()
# Копія екземпляра x
y=MyClass(x)
# Відображення полів екземпляра y
y.show()
print("Поля екземпляра x змінюються!")
# Зміна значення поля name
# екземпляра x
x.name="Java"
# Зміна значення елемента в списку
# nums - поле екземпляра x
x.nums[0]=0
print("Екземпляр x:")
# Відображення полів екземпляра x
x.show()
print("Екземпляр y:")
# Відображення полів екземпляра y
y.show()
```

Нижче наведено результат виконання цього програмного коду:



Результат виконання програми (з лістингу 6.16)

```
Екземпляр x:
name -> Python
nums -> [1, 2, 3]
name -> Python
nums -> [1, 2, 3]
Поля екземпляра x змінюються!
Екземпляр x:
name -> Java
```

```
nums -> [0, 2, 3]
```

Екземпляр у:

```
name -> Python
```

```
nums -> [1, 2, 3]
```

Тут в описі конструктора екземпляра класу `MyClass` ми передбачили можливість передати конструктору два аргументи (текст і список) або один аргумент, який є посиланням на вже існуючий екземпляр класу `MyClass` (для створення копії).



Для аргументу `nums` в описі конструктора задано значення за замовчуванням `None`, що відповідає порожньому посиланню. У певному сенсі це такий формальний прийом, оскільки зовсім не вказати значення за замовчуванням ми не могли. Якщо для аргументу `nums` не задати значення за замовчуванням, то під час виклику конструктора створення копії довелося б передавати і другий аргумент, який нічого не значить. А це не дуже зручно.

Узагалі, щоб оцінити різницю в підходах, хто бажає, може порівняти програмні коди в лістингах 6.14–6.16.

У конструкторі використано такий же прийом, що й у попередньому прикладі: перевіряється, чи збігається тип аргументу `arg` із класом `MyClass`. Спосіб обробки аргументів конструктора залежить від істинності або хибності цієї умови. Принципова відмінність, порівняно з попереднім прикладом, полягає в тому, що під час присвоювання значень полям виконується копіювання значень із використанням *зрізу* (інструкція `[:]` після імені змінної, яка посилається на текст або список). У цьому разі неявно припускається, що тип аргументів дозволяє виконувати таку операцію.



Нагадаємо, що за допомогою отримання зрізу створюється **поверхнева копія** списку. У читача можуть виникнути запитання щодо прикладів, розглянутих у лістингу 6.14 і лістингу 6.16. Спробуємо їх передбачити. Для цього детальніше розглянемо, що відбувається при створенні копії екземпляра (лістинг 6.14). Там за допомогою функції `copy()` створювалася поверхнева копія у екземпляра `x` класу `MyClass`. У цього екземпляра `x` було поле-список `nums`. При створенні нового екземпляра `y` в нього теж буде поле `nums`. Значення цього поля таке ж, як значення поля `nums` вихідного

екземпляра `x`. Але насправді значення поля вихідного екземпляра `x` — це посилання на список (у цьому разі `[1, 2, 3]`). Тому в новому екземплярі у поле `nums` буде посилятися на той самий список, на який посилається поле `nums` екземпляра `x`. Якщо ми через екземпляр `x` і його поле `nums` змінюємо елемент у відповідному списку, то зміни «помітить» й екземпляр `y` (оскільки його поле посилається на той же самий список). Але якщо ми присвоїмо інше значення полю `nums` екземпляра `x`, то значення поля `nums` екземпляра `y` залишиться незмінним! Зверніть увагу — йдеться про присвоювання значення полю `nums`, а не окремому елементу списку `nums`: хто має бажання, може в програмному коді з лістинга 6.14 команду `x_nums[0]=0` замінити на команду `x_nums=0` і перевірити результат виконання коду. Чому так відбувається? Тому що під час присвоювання значення полю `nums` екземпляра `x` посилання в цьому полі просто перекидається на нові дані. При цьому поле `nums` екземпляра `y` продовжує посилятися на список.

У лістингу 6.16 у конструкторі екземпляра класу при створенні копії полів `name` і `nums` використано процедуру отримання зрізу. У цьому випадку створюються **поверхневі копії** полів `name` і `nums` вихідного екземпляра, і посилання на ці копії записуються в поля `name` і `nums` екземпляра-копії. Що ж стосується поля `nums`, то йдеться про **поверхневу копію** списку. Але оскільки в цьому разі поле-список `nums` не містить як елементи інших списків, проблем не виникає. Вони могли б бути, якщо б у вихідному екземплярі `x` список `nums` містив внутрішні списки. Наприклад, якщо в лістингу 6.16 команду `x=MyClass("Python", [1, 2, 3])` замінити на команду `x=MyClass("Python", [[1, 2], 3])`, а команду `x_nums[0]=0` — на команду `x_nums[0][0]=0`, то зміни в полі `nums` екземпляра `x` відобразяться й на полі `nums` екземпляра `y`.

Резюме

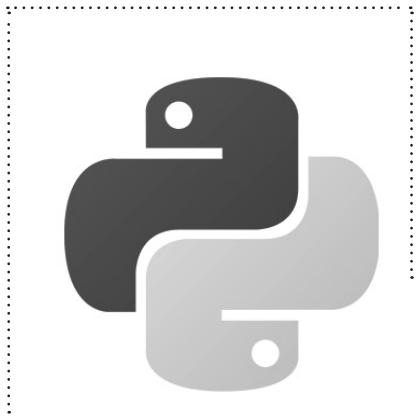
Я би і свідомість утратив, та ви мені
і так повірите.

З телесеріалу «Альф»

- У мові Python підтримується парадигма об'єктно-орієнтованого програмування (ООП). Програмні коди, написані мовою Python, можуть містити й оперувати класами й екземплярами класів.
- У загальному сенсі клас — це деякий шаблон, за яким створюються об'єкти. Об'єкт, у свою чергу, об'єднує в одне щіле дані й функції (або методи) для обробки цих даних.
- Об'єкти, створювані на основі класу, називають екземплярами класу. Сам клас у Python також є об'єктом.
- Змінні, оголошені в класі, називаються полями класу. Змінні, пов'язані з екземплярами класу, називаються полями екземпляра класу. Функції, описані в класі, називаються функціями класу. Функції, пов'язані з екземплярами класу, називаються методами екземпляра класу. Поля й методи (функції) називаються атрибутами (класу або екземпляра класу відповідно).
- Опис класу починається з ключового слова `class`, після якого вказується ім'я класу. Методи екземпляра класу описуються в тілі класу як звичайні функції. При цьому перший аргумент методу (рекомендована назва `self`) за замовчуванням є посиланням на екземпляр, із якого викликається метод. Під час виклику методу з екземпляра класу після імені екземпляра через крапку вказується ім'я методу, а в круглих дужках — аргументи. Посилання на екземпляр класу, із якого викликається метод, передається автоматично, так що під час виклику

методу в нього на один аргумент менше, ніж це було під час опису методу в тілі класу.

- Для створення екземпляра класу змінній екземпляра присвоюється вираз, що складається з імені класу і круглих дужок. У круглих дужках можуть передаватися аргументи конструктору екземпляра класу.
- Конструктор екземпляра класу — це метод, який автоматично викликається при створенні екземпляра класу. Конструктор має назву `__init__()`. Також існує деструктор `__del__()`, але він на практиці використовується рідко.
- Атрибути об'єкта класу й екземпляра класу вже після їхнього створення можна змінювати (додавати й видаляти). Для видалення атрибутів можна використовувати оператор `del`. Додавання атрибутів виконується шляхом присвоювання їм значення.
- Якщо в класу й в екземпляра класу є поля з однаковими назвами, поле екземпляра класу «перекриває» поле класу. Звертання до такого поля за іменем (через посилання на екземпляр класу) буде означати звертання до поля екземпляра класу.
- Під час звертання до функції, описаної в тілі класу, через посилання на екземпляр класу, створюється об'єкт методу екземпляра класу, який і викликається. Виклик цього методу еквівалентний виклику відповідної функції класу з такими ж аргументами, що й у методу, але ще з додаванням додаткового першого аргументу — посилання на екземпляр класу, з якого викликається метод.
- Ім'я функції, описаної в класі, є атрибутом цього класу. Як будь-якому атрибуту, цьому атрибуту може бути присвоєно нове значення (ім'я функції), його може бути видалено, або в клас може бути додано новий атрибут, що посилається на функцію.
- Посилання на функцію класу або на метод екземпляра класу може бути записано в змінну, яка потім використовується для виклику функції або методу.
- Для створення копії екземпляра класу описують конструктор створення копії або використовують функції `copy()` і `deepcopy()` із модуля `copy`.



Розділ 7

Продовжуємо зайомство з ООП

Сидів собі тихо, мирно. Потім згоднів. Далі, як в тумані.

З телесеріалу «Альф»

Далі йтиметься про деякі досить нетривіальні механізми, що є невід'ємною частиною ООП. Насамперед, це, звичайно ж, наслідування. Крім наслідування, ми обговоримо використання спеціальних методів й атрибутів при роботі з класами, познайомимося з можливостями щодо перевантаження операторів, а також торкнемося низки інших тем.

Наслідування

Боюся, тобі доведеться любити
мене до скону.

З телесеріалу «Альф»

Якщо в загальних рисах, то ідея *наслідування* полягає в тому, що новий клас створюється не на «порожньому місці», а на основі вже існуючого класу. Припустімо, що один клас створюється на основі іншого класу. Той клас, що створюється, будемо називати *похідним класом*. Той клас, на основі якого створюється похідний клас, називатимемо *базовим класом*. Таким чином, за визначенням, похідний клас створюється на основі базового класу. В результаті наслідування всі поля й функції з базового класу є неявно «наслідувальними» в похідному класі. З формальної точки зору, щоб створити похідний клас на основі базового, в описі похідного класу після імені класу в круглих дужках указується ім'я базового класу. Іншими словами, описуючи похідний клас, використовуємо такий шаблон (жирним шрифтом виділено ключові елементи шаблону):

```
class похідний_клас(базовий_клас) :  
    # тіло класу
```

Приклад наслідування проілюстровано програмним кодом у лістингу 7.1.

Лістинг 7.1. Наслідування класів

```
# Базовий клас  
class BaseClass:  
    # Поле базового класу  
    name_base="Клас BaseClass"
```

Phyton

```
# Метод екземпляра базового класу
def say_base(self):
    print("Метод say_base()")

# Похідний клас
class NewClass(BaseClass):
    # Поле похідного класу
    name_new="Клас NewClass"
    # Метод екземпляра похідного класу
    def say_new(self):
        print("Метод say_new()")

# Екземпляр базового класу
obj_base=BaseClass()
# Екземпляр похідного класу
obj_new=NewClass()
print("Клас BaseClass і екземпляр obj_base:")
# Поле базового класу
print(BaseClass.name_base)
# Метод екземпляра базового класу
obj_base.say_base()
print("\nКлас NewClass і екземпляр obj_new:")
# Поле похідного класу
print(NewClass.name_base)
# Метод екземпляра похідного класу
obj_new.say_base()
# Наслідуване з базового класу
# поле похідного класу
print(NewClass.name_new)
# Наслідуваний із базового класу
# метод екземпляра похідного класу
obj_new.say_new()
```

У програмному коді базовий клас називається `BaseClass`. Це — звичайний клас. Обставина, що цей клас слугує базовим для створення нового класу, ніяк не відображеня в класі `BaseClass`. У класі `BaseClass` описано поле `name_base` з текстовим значенням "Клас BaseClass" і метод екземпляра з назвою `say_base()`. У методу один аргумент, і метод у вікні виводу відображує просте повідомлення.

Щодо похідного класу, то він називається NewClass. Під час створення цього класу в круглих дужках після імені створюваного похідного класу вказується ім'я базового класу BaseClass. Безпосередньо в тілі класу NewClass описано поле name_new із текстовим значенням "Клас NewClass" і призначений для виведення повідомлення метод екземпляра класу say_new() із одним аргументом. Але оскільки клас NewClass створюється на основі класу BaseClass, то «у спадок» від цього класу він отримує поле name_base і метод say_base(). Отже, у класу NewClass два поля (name_new і name_base) і два методи (say_new() і say_base()).

Екземпляр базового класу створюємо командою obj_base=BaseClass(), а екземпляр похідного класу — командою obj_new>NewClass(). Через клас BaseClass отримуємо доступ до поля name_base. Через екземпляр obj_base базового класу викликаємо метод say_base(). У класу NewClass, як зазначалося, є поля name_base і name_new, у екземпляра obj_new — методи say_base() і say_new(), які, власне, і викликаються. Результат виконання описаного програмного коду наведено нижче:

Результат виконання програми (з лістингу 7.1)

Клас BaseClass і екземпляр obj_base:

Клас BaseClass

Метод say_base()

Клас NewClass і екземпляр obj_new:

Клас BaseClass

Метод say_base()

Клас NewClass

Метод say_new()

Важливо розуміти, що при створенні похідного класу на основі базового класу останній використовується не для того, аби просто створити на основі його атрибутів такі ж атрибути в похідному класі: у похідному класі використовуються ті самі атрибути, що й у базовому класі. Тому, якщо вже після створення похідного класу змінити «вміст» базового класу, ці зміни відобразяться і на похідному класі. Приклад такої ситуації проілюстровано в лістингу 7.2.

Лістинг 7.2. Зміна базового класу

```
# Базовий клас
class BaseClass:
    # Поле базового класу
    name="Поле name"
    # Метод екземпляра базового класу
    def say(self):
        print("Метод say()")
# Похідний клас
class NewClass(BaseClass):
    pass
# Екземпляр похідного класу
obj=NewClass()
# Поле похідного класу
print(NewClass.name)
# Метод екземпляра похідного класу
obj.say()
# Створюємо функцію
def hello(self):
    print("Новий метод hello()")
# Змінюємо посилання на функцію
# в базовому класі
BaseClass.say=hello
# Змінюємо значення поля
# в базовому класі
BaseClass.name="Нове значення поля name"
# Перевіряємо значення поля в похідному класі
print(NewClass.name)
# Викликаємо метод з екземпляра
# похідного класу
obj.say()
```

Результат виконання цього програмного коду наведено нижче:

Результат виконання програми (з листингу 7.2)

```
Поле name
Метод say()
Нове значення поля name
Новий метод hello()
```

У базовому класі `BaseClass` описано поле `name` з ізначенням "Поле `name`" і метод `say()`, який відображає у вікні виводу текст "Метод `say()`". Похідний клас `NewClass`, який створюється на основі базового класу `BaseClass`, жодного додаткового коду не містить. Ми створюємо екземпляр `obj` похідного класу `NewClass`. Командами `print(NewClass.name)` і `obj.say()` перевіряємо, що поле `name` і метод `say()` успішно наслідувалися з базового класу. Потім визначається функція `hello()`, і за допомогою команди `BaseClass.say=hello` посилання на цю функцію записується в атрибут `say` базового класу `BaseClass`. Також за допомогою команди `BaseClass.name="Нове значення поля name"` змінюємо значення поля `name` в базовому класі. Після цього перевіряємо значення поля `name` в похідному класі за допомогою команди `print(NewClass.name)`, а також викликаємо метод `say()` із екземпляра похідного класу за допомогою команди `obj.say()`. Як видно з результату виконання цих команд, у похідному класі використано нові значення атрибутів базового класу.

Під час наслідування існує можливість *перевизначати* поля й методи, наслідувані в похідному класі з базового. Загальна ідея полягає в тому, що поле або метод, які наслідуються з базового класу, у похідному класі створюються заново. Як це відбувається на практиці, ілюструє програмний код із листингу 7.3.

Листинг 7.3. Перевизначення полів і методів

```
# Базовий клас
class BaseClass:
    # Поле базового класу
    name="Поле name базового класу"
    # Метод екземпляра базового класу
    def say(self):
        print("Метод say() базового класу")
# Похідний клас
```

Phyton

```
class NewClass(BaseClass):
    # Поле похідного класу
    name="Поле name похідного класу"
    # Метод екземпляра похідного класу
    def say(self):
        print("Метод say() похідного класу")
# Екземпляр базового класу
obj_base=BaseClass()
# Екземпляр похідного класу
obj_new=NewClass()
# Атрибути екземпляра базового класу
print(obj_base.name)
obj_base.say()
# Атрибути екземпляра похідного класу
print(obj_new.name)
obj_new.say()
```

При виконанні цього програмного коду отримуємо такий результат:



Результат виконання програми (з лістингу 7.3)

```
Поле name базового класу
Метод say() базового класу
Поле name похідного класу
Метод say() похідного класу
```

У цьому прикладі ми спочатку створили базовий клас `BaseClass` із полем `name` і методом `say()`, а потім на основі цього класу створили похідний клас `NewClass`. І хоча під час наслідування поле `name` і метод `say()` стають доступними і в похідному класі, ми все одно в класі `NewClass` описуємо поле `name` і метод `say()`, але вже інші (порівняно з базовим класом). У результаті екземпляр `obj_base` базового класу посилається на поле і метод із базового класу, а екземпляр `obj_new` похідного класу посилається на поле і метод похідного класу.

Зазвичай на практиці вдаються до перевизначення методів. Причому ситуація може бути далеко не такою простою, як було показано вище.

Ще один приклад перевизначення методів під час наслідування подано в лістингу 7.4.

Лістинг 7.4. Перевизначення методів

```
# Базовий клас
class BaseClass:
    def __init__(self, num):
        self.id=num
    def get(self):
        print("ID:",self.id)
    # Метод екземпляра базового класу
    def show(self):
        print("Поле екземпляра базового класу")
        self.get()
# Похідний клас
class NewClass(BaseClass):
    def __init__(self,num,txt):
        super().__init__(num)
        self.name=txt
    def get(self):
        super().get()
        print("Name:",self.name)
# Екземпляр базового класу
obj_base=BaseClass(1)
print("Викликаємо метод show() із екземпляра obj_base:")
# Виклик методу екземпляра базового класу
obj_base.show()
# Екземпляр похідного класу
obj_new=NewClass(10,"десятка")
print("Викликаємо метод show() із екземпляра obj_new:")
# Виклик методу екземпляра похідного класу
obj_new.show()
```

Перш ніж подивитися, який же буде результат виконання цього програмного коду, коротко проаналізуємо його.

Базовий клас `BaseClass` містить конструктор, у якому полю `id` екземпляра класу присвоюється значення (передається аргументом конструктору). Метод екземпляра `get()` відображає у вікні виводу значення поля `id` екземпляра, з якого викликається метод. Ще в класі `BaseClass` описано метод екземпляра `show()`, у якому за допомогою команди `print("Поле екземпляра базового класу")` у вікні виводу відображується повідомлення, а потім за допомогою команди `self.get()` викликається метод екземпляра `get()` (який, нагадаємо, відображає значення поля `id` екземпляра класу).

Похідний клас `NewClass` створюється на основі базового класу `BaseClass`. У цьому класі багато нових (і, можливо, незрозумілих) для нас інструкцій. Насамперед, це конструктор. Ми припускаємо, що в екземпляра похідного класу буде два поля: поле `id` і поле `name`. Необхідно, щоб цим полям у конструкторі присвоювалися значення, і, відповідно, значення цих полів повинні передаватися як аргументи конструктору. Наша ідея полягає в тому, щоб у конструкторі екземпляра похідного класу викликати конструктор екземпляра базового класу. Проблема поглибується тим, що всі конструктори у всіх класах називаються однаково. Щоб викликати «правильний» конструктор, нам знадобиться функція `super()`. Вона повертає результатом спеціальний проксі-об'єкт, який делегує виклик методу базовому класу. З практичної точки зору, цю функцію можна розглядати як замісник екземпляра базового класу, із якого вдається викликати вихідну (визначену в базовому класі) версію методу — у тому числі, що стосується й конструктора.



Як би там не було, а викликати конструктор базового класу можна за допомогою інструкції вигляду `super().__init__(аргументи)`. У круглих дужках для конструктора передаються аргументи, причому без посилання на екземпляр класу, з якого викликається конструктор.

За допомогою команди `super().__init__(num)` у тілі конструктора екземпляра похідного класу викликається конструктор екземпляра базового класу. У результаті поле `id` отримує своє значення. Потім командою `self.name = txt` присвоюється значення полю `name` екземпляра похідного класу (zmінні `self`, `num` і `txt` позначають аргументи конструктора екземпляра похідного класу).

У похідному класі перевизначається метод `get()`. У тілі методу команда `super().get()` викликає версію методу з базового класу. При виконанні цього методу, нагадаємо, у вікні виводу відображується значення поля `id` екземпляра класу, з якого викликається метод. Потім за допомогою команди `print("Name:", self.name)` відображується значення поля `name` похідного класу.



Якщо є деякий метод, визначений в базовому класі й перевизначений в похідному класі, і нам необхідно в тілі екземпляра похідного класу викликати цю вихідну, базову версію методу (а таке можливо), використовуємо інструкцію вигляду `super().метод(аргументи)`.

Є альтернатива використанню функції `super()`. Наприклад, якщо нам у похідному класі треба викликати версію методу екземпляра з базового класу, можемо виконати явне посилання на базовий клас, скориставшись командою `вигляду базовий_клас.метод(self, аргументи)`, — це замість команди `super().метод(аргументи)`. Наприклад, команда `super().get()` могла б виглядати як `BaseClass.get(self)`.

Важливий момент, на якому варто зупинитися, пов'язаний із методом `show()`. Він описаний у базовому класі, а в похідному не перевизначений. Тому похідний клас наслідує версію методу `show()` із базового класу. Але в методі `show()` (у базовому класі) викликається метод `get()`, який, у свою чергу, перевизначається в похідному класі. Під час виклику методу `show()` із екземпляра похідного класу буде викликатися метод `get()`. Питання в тому, яким саме буде цей метод `get()` — із базового класу чи з похідного? Відповідь на це питання ми отримуємо з результату виконання програмного коду.

Екземпляр базового класу створюємо командою `obj_base=BaseClass(1)`. Потім викликаємо метод `show()` із екземпляра `obj_base` базового класу (команда `obj_base.show()`).

Екземпляр похідного класу створюємо командою `obj_new=NewClass(10, "десятка")`. Командою `obj_new.show()` викликається метод `show()` із екземпляра `obj_new` похідного класу. Результат виконання всього програмного коду такий:

 Результат виконання програми (з лістингу 7.4)

Викликаємо метод `show()` із екземпляра `obj_base`:

Поле екземпляра базового класу

ID: 1

Викликаємо метод `show()` із екземпляра `obj_new`:

Поле екземпляра базового класу

ID: 10

Name: десятка

Особливий інтерес викликають останні три рядки у вікні виводу: це результат виконання команди `obj_new.show()`. Нескладно зрозуміти, що такий результат можна отримати, якщо в тілі наслідуваного з базового класу методу `show()` викликається перевизначений у похідному класі метод `get()`. Причому це загальне правило: якщо в наслідуваному методі є команда виклику іншого методу, і цей інший метод перевизначається в похідному класі, то буде викликано цю нову (перевизначену) версію методу.



У мові C++ така властивість називається **віртуальністю**. Висловлюючись термінологією мови C++, у мові Python усі методи є **віртуальними**.

До цього при створенні похідного класу базовий клас був завжди один. Насправді в Python похідний клас можна створювати на основі одразу декількох базових класів. Це називається **множинним наслідуванням**. При множинному наслідуванні у похідного класу не один базовий клас, а декілька базових класів. Похідний клас наслідує всі атрибути кожного зі своїх базових класів. Невеликий приклад із множинним наслідуванням наведено в лістингу 7.5.



У цьому прикладі ми наче умовно «описуємо» деяку коробку або ящик. У цього ящика є геометричні розміри (ширина, висота, глибина), маса (або вага — хоча, якщо чесно, це далеко не одне й те саме), а ще колір. На основі лінійних розмірів обчислюється об'єм (добуток ширини, висоти й глибини). Усі одиниці вимірювань безрозмірні.

У класі BoxSize передбачено можливість записувати лінійні розміри й обчислювати об'єм. Клас BoxParams передбачає можливість запам'ятувати такі параметри, як вага і колір. На основі класів BoxSize і BoxParams через наслідування створюється клас Box.

Лістинг 7.5. Наслідування декількох базових класів

```
# Перший базовий клас
class BoxSize:
    # Конструктор
    def __init__(self, width, height, depth):
        # Присвоювання значень полям екземпляра
        self.width=width
        self.height=height
        self.depth=depth
    # Метод для обчислення об'єму
    def volume(self):
        # Результат - добуток полів екземпляра
        return self.width*self.height*self.depth
    # Метод для відображення значень полів екземпляра
    # і результату виклику методу volume()
    def show(self):
        # Поля екземпляра класу
        print("Розміри й об'єм ящика:")
        print("Ширина:", self.width)
        print("Висота:", self.height)
        print("Глибина:", self.depth)
        # Результат виклику методу volume()
        print("Об'єм:", self.volume())
# Другий базовий клас
class BoxParams:
    # Конструктор
    def __init__(self, weight, color):
        # Присвоювання значень полям екземпляра
        self.weight=weight
        self.color=color
    # Метод для відображення значень полів екземпляра
    def show(self):
```

```

# Відображення значень полів
print("Додаткові параметри ящика:")
print("Вага (маса):",self.weight)
print("Колір:",self.color)

# Похідний клас
class Box(BoxSize,BoxParams):
    # Конструктор
    def __init__(self,width,height,depth,weight,color):
        # Виклик конструктора першого базового класу
        BoxSize.__init__(self,weight,height,depth)
        # Виклик конструктора другого базового класу
        BoxParams.__init__(self,weight,color)
        # Виклики методу show() екземпляра класу
        self.show()

    # Перевизначення методу show()
    def show(self):
        # Виклик методу show() із першого базового класу
        BoxSize.show(self)
        # Виклик методу show() із другого базового класу
        BoxParams.show(self)

# Створюємо екземпляр похідного класу
obj=Box(10,20,30,5,"зелений")

```

У базовому класі BoxSize описано конструктор, у якому полям width, height і depth екземпляра класу присвоюються значення (тим самим при створенні екземпляра класу цьому екземпляру додаються відповідні поля).



Зверніть увагу, що аргументи конструктора width, height, depth і поля екземпляра класу називаються однаково. Якщо в тілі конструктора ми просто пишемо назви width, height або depth, то маємо на увазі аргументи конструктора. Для посилання на поля екземпляра класу використовуємо інструкції self.width, self.height і self.depth із явним зазначенням аргументу конструктора self, який ототожнюється з посиланням на екземпляр класу.

Для тих, хто знайомий із мовами C++, Java або C#, підкреслимо: у мові Python немає скороченої форми звертання до полів екземпляра класу.

Також у класі `BoxSize` описується метод `volume()`, призначений для обчислення об'єму: результатом методу є добуток `self.width*self.height*self.depth` значень усіх трьох полів екземпляра.

Метод `show()` містить команди з відображення у вікні виводу значень полів екземпляра й результату виклику методу `volume()`.

У базовому класі `BoxParams` у конструкторі задаються значення двох полів, а метод `show()` у цьому класі описано так, що під час його виклику відображуються значення цих полів.

Похідний клас `Box` створюється на основі базових класів `BoxSize` і `BoxParams`: назви цих класів указані в круглих дужках після імені похідного класу. У похідному класі описується конструктор, у якого досить багато аргументів. У тілі конструктора послідовно викликаються конструктори базових класів: командою `BoxSize.__init__(self,-weight,height,depth)` викликається конструктор екземпляра класу `BoxSize`, а командою `BoxParams.__init__(self,weight,color)` викликається конструктор екземпляра класу `BoxParams`. І в тому, і в іншому випадках конструктор викликається як функція класу з явним за-значенням імені класу й передачею першим аргументом конструктору посилання `self` на екземпляр класу. Також у конструкторі викликається метод `show()`. Але тут ми теоретично повинні були б зустрітися з проблемою: у кожному з базових класів `BoxSize` і `BoxParams` є метод із назовою `show()`, і кожний із цих методів наслідується в похідному класі `Box`. У принципі, в такій ситуації за замовчуванням при звертанні до методу `show()` викликалася б версія методу з класу `BoxSize`, оскільки в списку базових класів цей клас зазначено першим.



В екземплярі похідного класу при звертанні до поля або методу, якщо це поле або метод не описані в тілі похідного класу, пошук починається в базових класах, причому строго відповідно до того порядку, в якому класи вказано в списку наслідування (у круглих дужках після імені похідного класу).

Але ми йдемо іншим шляхом — шляхом перевизначення методу `show()` у похідному класі `Box`. У тілі методу в класі `Box` командами `BoxSize.show(self)` і `BoxParams.show(self)` послідовно викликаються версії цього методу відповідно з класу `BoxSize` і `BoxParams`. Тут діє той же принцип, що й під час перевизначення конструктора класу `Box`: методи викликаються як функції класу через посилання на об'єкт класу і їм аргументами передаються посилання на екземпляр класу.

Окрім описання класів, у програмному коді є лише одна команда `obj=Box(10, 20, 30, 5, "зелений")`, якою створюється екземпляр похідного класу `Box`. При цьому викликається конструктор, а в конструкторі викликається метод `show()`, який відображує значення всіх полів екземпляра `obj`. Отже, результат отримуємо такий:



Результат виконання програми (з лістингу 7.5)

Розміри й об'єм ящика:

Ширина: 5

Висота: 20

Глибина: 30

Об'єм: 3000

Додаткові параметри ящика:

Вага (маса): 5

Колір: зелений

Окрім множинного наслідування може використовуватися **багатократне наслідування**. При багатократному наслідуванні похідний клас є базовим для іншого класу. Якщо врахувати, що одночасно допустимо використовувати і множинне наслідування, стає очевидним, що структура «родинних стосунків» між класами може бути дуже складною.



Часто все зводиться до визначення того, як під час багатократного й одночасно множинного наслідування похідним класом наслідуються з різних класів атрибути (методи/функції) з однаковими назвами. Ідеється ось про що: уявімо, що з екземпляра похідного класу викликається метод або з похідного класу викликається функція, а в структурі наслідування класів є методи/функції з таким же ім'ям. Питання в тому, яка версія (із якого класу) методу або функції буде викликатися. У загальному, спрощеному вигляді відповідь

на це питання зводиться до того, що послідовно перебирається ланцюжок наслідування класів до першого «збігу». Базові класи перебираються в тому порядку, як їх вказано під час створення похідного класу. Причому базові класи «переглядаються» разом зі своїми базовими класами. Хоча й тут не так усе просто. Неприємності виникають тоді, коли один і той самий клас опиняється «різними шляхами», пряма або опосередкована, декілька разів серед наслідуваних класів. Наприклад, клас `A` є базовим для класів `B` і `C`, а на їх основі створюється клас `D`. У результаті виходить, що клас `D` наслідує клас `A` «двічі»: через клас `B` і через клас `C`. А можуть бути й більш трагічні «хитросплетіння». У подібних випадках у Python застосовується спеціальний алгоритм, за яким визначається ланцюжок наслідування класів, причому в цьому ланцюжку кожен клас зустрічається лише раз.

Ми з цього приводу розглянемо невеликий приклад з лістингу 7.6. У наведеному там програмному коді описано декілька класів, причому одні з них наслідують інші. Загальну схему наслідування класів наведено на рис. 7.1.

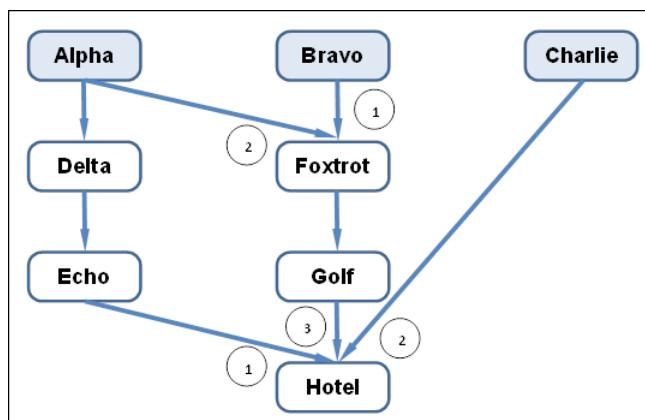


Рис. 7.1. Схема наслідування класів. Наслідування зображене напрямленою стрілкою (від базового класу до похідного). Цифрами позначені порядок наслідування класів у випадку множинного наслідування. Класи з описаною функцією `hi()` затемнено

Три базових класи (`Alpha`, `Bravo` і `Charlie`) містять опис функції `hi()`, а інші класи наслідують (різними шляхами) ці класи. Програмний код такий:

Phyton

Лістинг 7.6. Багатократне наслідування

```
# Описуємо класи
class Alpha:
    def hi():
        print("Клас Alpha")
class Bravo:
    def hi():
        print("Клас Bravo")
class Charlie:
    def hi():
        print("Клас Charlie")
class Delta(Alpha):
    pass
class Echo(Delta):
    pass
class Foxtrot(Bravo,Alpha):
    pass
class Golf(Foxtrot):
    pass
class Hotel(Echo,Charlie,Golf):
    pass
# Викликаємо функції класів
Echo.hi()
Golf.hi()
Hotel.hi()
```

Таким буде результат виконання програмного коду:

Результат виконання програми (з лістингу 7.6)

Клас Alpha

Клас Bravo

Клас Charlie

Після опису класів у програмному коді послідовно виконуються команди Echo.hi(), Golf.hi() і Hotel.hi(). Розберімо, що відбувається при виконанні кожної з цих команд.

Під час виконання команди `Echo.hi()` починається пошук функції `hi()` у тілі класу `Echo`. Безпосередньо в цьому класі функцію не описано. Але клас `Echo` створюється на основі класу `Delta`. У цьому класі функцію з іменем `hi()` також не описано, тому пошук функції виконується в класі `Alpha` — базовому класі для класу `Delta`. У результаті викликається функція з класу `Alpha`, і у вікні виводу з'являється повідомлення Клас `Alpha`. Тут ситуація проста, оскільки простий ланцюжок наслідування: клас `Echo` наслідує клас `Delta`, а клас `Delta` наслідує клас `Alpha`. Пошук функції `hi()` здійснюється відповідно до цього ланцюжка наслідування.

Під час виконання команди `Golf.hi()` пошук функції `hi()` спочатку починається в тілі класу `Golf`. Але там така функція не описана. Оскільки клас `Golf` створювався на основі класу `Foxtrot`, пошук функції `hi()` виконується в цьому класі. У тілі класу `Foxtrot` функцію `hi()` також не описано. Пошук продовжується в базових класах для класу `Foxtrot`: це класи `Bravo` й `Alpha` (у тому порядку, як їх указано під час наслідування). Оскільки в класі `Bravo` є опис функції `hi()`, вона й викликається, в результаті чого у вікні виводу з'являється повідомлення Клас `Bravo`. Якщо б у класі `Bravo` функції `hi()` не виявилося, пошук би продовжився в класі `Alpha`.

Складніша ситуація з командою `Hotel.hi()`. За логікою, для пошуку функції `hi()` класи повинні були б перевірятися в такій послідовності: спочатку `Hotel`, `Echo`, `Delta`, `Alpha`, потім `Charlie`, а далі `Golf`, `Foxtrot`, `Bravo` й `Alpha` — поки не буде знайдено перший клас у цій послідовності, в якому описано функцію `hi()`. У цьому ланцюжку перший клас, у якому описано функцію `hi()`, — клас `Alpha`. Тому, якщо б усе так і було, з'явилося б повідомлення Клас `Alpha`. Але насправді з'являється повідомлення Клас `Charlie`. Чому так? Насправді, повна й вичерпна відповідь зводилася б до опису алгоритму, який використовується у Python при формуванні ланцюжка наслідування, коли одні й ті самі класи багатократно наслідуються в похідному класі. Нам такі «технічні» подробиці навряд чи знадобляться, тому ми обмежимося загальною ідеєю з апеляцією до нашого прикладу. Отже, у формальній послідовності наслідуваних класів клас `Alpha` зустрічається двічі. А треба, щоб він там зустрічався всього один раз. Причому важлива позиція, на якій цей клас знаходитиметься в ланцюжку наслідування (оскільки вона визначає, в якій послідовності переглядаються класи під час пошуку функції

hi()). Базовий принцип полягає в тому, щоб уникати ситуацій, коли спочатку переглядається базовий клас, а потім похідний від нього. У цьому конкретному випадку теоретично є два варіанти ланцюжка наслідування для класу Hotel: Echo, Delta, Alpha, Charlie, Golf, Foxtrot, Bravo або Echo, Delta, Charlie, Golf, Foxtrot, Bravo, Alpha. У першому випадку клас Alpha переглядається перед класом Foxtrot, для якого клас Alpha є базовим. Як ми згадували вище, це поганий варіант. У другому випадку такої проблеми немає. Тому ланцюжок наслідування класів для класу Hotel насправді виглядає так: Echo, Delta, Charlie, Golf, Foxtrot, Bravo, Alpha. Рухаючись цим ланцюжком під час пошуку коду функції hi() (ідеться, нагадаємо, про виконання команди Hotel.hi()), «зупиняємося» на класі Charlie, в якому цю функцію описано.



Аби побачити «ланцюжок наслідування» для похідного класу, розумно скористатися полем `__mro__` цього класу. Зокрема, якщо в розглянутому вище прикладі додати команду `print(Hotel.__mro__)`, побачимо, в якій послідовності наслідуються класи в класі Hotel.

Узагалі, нескладно придумати таку схему наслідування класів, яка є неприпустимою в тому сенсі, що не дозволяє встановити послідовність або ланцюжок наслідування класів. Наприклад, клас C наслідує класи A і B, клас D наслідує класи B і A, а клас E наслідує класи C і D. У такій ситуації виникає помилка типу `TypeError`, пов'язана з неможливістю встановити ланцюжок наслідування.

У наступному пункті йтиметься про деякі *спеціальні методи та поля*, які дозволяють суттєво розширити можливості класів та екземплярів класів у мові Python.

Спеціальні методи і поля

Це я такий чутливий, чи в кімнаті
якась напруженість?

З телесеріалу «Альф»

У Python є група методів, назви яких починаються й закінчуються по-двоїним підкresлованням. Такі методи призначені для роботи з класами й екземплярами класів і дозволяють виконувати деякі дуже спеціфічні операції. Тому ці методи зазвичай називають *спеціальними*.



Принаймні з двома спеціальними методами ми вже знайомі: це конструктор `__init__()` і деструктор `__del__()`.

Також є група полів, які в назві на початку й у кінці містять по два символи підкresловання. Ці поля доступні для кожного класу й/або екземпляра за замовчуванням.



На перший погляд, може здатися дивним, що у класів, які створюються користувачем, є атрибути, які користувачем не описані. Але тут варто врахувати, що починаючи з версії Python 3 усі класи (в тому числі й створені користувачем), зрештою, є нашадками класу `object`. Іншими словами, існує ієрархія класів, і на вершині цієї ієрархії знаходиться клас `object`.

У таблиці 7.1 перераховано деякі спеціальні поля, які можуть бути корисними під час роботи з класами, а також їхні призначення.

Таблиця 7.1. Спеціальні поля

Поле	Призначення
<code>__bases__</code>	Повертається список базових класів
<code>__dict__</code>	Повертається словник із атрибутами класу
<code>__doc__</code>	Повертається текст документування класу (текст, що описує клас). Текст документування можна безпосередньо вказати в тілі класу першим рядком або присвоїти полю значення
<code>__module__</code>	Повертається модуль класу
<code>__mro__</code>	Повертається ланцюжок наслідування класу
<code>__name__</code>	Повертається ім'я класу
<code>__qualname__</code>	Повертається повне ім'я класу (у «крапковому» форматі, що відображає структуру вкладених класів)



В екземплярах класу є спеціальне поле `__class__`, яке дозволяє визначити клас, на основі якого створюється екземпляр. Оскільки клас сам є об'єктом, то у класу також є. Поле `__class__` для об'єкта класу дає (під час виведення результату функцією `print()`) значення `<class 'type'>`, що означає належність об'єкта класу до типу `type`.

Також є дуже корисною функція `dir()`, яка дозволяє отримати список атрибутів (полів і методів) екземпляра класу. Під час виклику функції екземпляр класу передається аргументом функції.

Невеликий приклад використання перерахованих полів наведено в лістингу 7.7.

Лістинг 7.7. Спеціальні поля

```
# Клас
class Alpha:
    """Клас Alpha і внутрішній клас Bravo"""
    def hello():
        pass
```

```

# Внутрішній клас
class Bravo:
    pass

# Похідний від Alpha клас
class Charlie(Alpha):
    pass

# Похідний від Charlie клас
class Delta(Charlie):
    pass

# Створюємо екземпляр класу
obj=Alpha()

# Поле __class__ екземпляра класу
print("Поле __class__ ")
print("Екземпляр obj:",obj.__class__)

# Поле __class__ класу
print("Клас Alpha:",Alpha.__class__)
print("Клас Alpha.Bravo:",Alpha.Bravo.__class__)
print("Клас Charlie:",Charlie.__class__)

# Поля __bases__ і __mro__
print("\nПоля __bases__ і __mro__")
print("Клас Delta, поле __bases__:",Delta.__bases__)
print("Клас Delta, поле __mro__:",Delta.__mro__)
print("Клас Alpha, поле __bases__:",Alpha.__bases__)
print("Клас Alpha, поле __mro__:",Alpha.__mro__)

# Поле __doc__
print("\nПоле __doc__")
print("Опис класу Alpha:",Alpha.__doc__)
Delta.__doc__="Клас Delta наслідує клас Charlie"
print("Опис класу Delta:",Delta.__doc__)

# Поле __module__
print("\nПоле __module__")
print("Модуль класу Alpha:",Alpha.__module__)

# Поле __dict__
print("\nПоле __dict__")
print("Атрибути класу Alpha:",Alpha.__dict__)
print("Атрибути класу Alpha.Bravo:",Alpha.Bravo.__dict__)
print("Атрибути класу Delta:",Delta.__dict__)

```

Phyton

```
# Поля __name__ і __qualname__
print("\nПоля __name__ і __qualname__")
print("Клас Alpha, поле __name__:",Alpha.__name__)
print("Клас Alpha, поле __qualname__:",Alpha.__qualname__)
print("Клас Alpha.Bravo, поле __name__:",Alpha.Bravo.__name__)
print("Клас Alpha.Bravo, поле __qualname__:",Alpha.Bravo.__qualname__)
print("Клас Delta, поле __name__:",Delta.__name__)
print("Клас Delta, поле __qualname__:",Delta.__qualname__)
```

Результат виконання цього програмного коду такий:

■ Результат виконання програми (з лістингу 7.7)

Поле __class__

Екземпляр obj: <class '__main__.Alpha'>

Клас Alpha: <class 'type'>

Клас Alpha.Bravo: <class 'type'>

Клас Charlie: <class 'type'>

Поля __bases__ і __mro__

Клас Delta, поле __bases__: (<class '__main__.Charlie'>,)

Клас Delta, поле __mro__: (<class '__main__.Delta'>,
<class '__main__.Charlie'>, <class '__main__.Alpha'>,
<class 'object'>)

Клас Alpha, поле __bases__: (<class 'object'>,)

Клас Alpha, поле __mro__: (<class '__main__.Alpha'>,
<class 'object'>)

Поле __doc__

Опис класу Alpha: Клас Alpha і внутрішній клас Bravo

Опис класу Delta: Клас Delta наслідує клас Charlie

Поле __module__

Модуль класу Alpha: __main__

Поле __dict__

Атрибути класу Alpha: {`'__module__': '__main__', '__doc__': 'Клас Alpha і внутрішній клас Bravo', 'hello': <function Alpha.hello at 0x01FF2540>, 'Bravo': <class '__main__.Alpha.Bravo'>, '__dict__': <attribute '__dict__' of 'Alpha' objects>, '__weakref__': <attribute '__weakref__' of 'Alpha' objects>}`

Атрибути класу Alpha.Bravo: {`'__module__': '__main__', '__dict__': <attribute '__dict__' of 'Bravo' objects>, '__weakref__': <attribute '__weakref__' of 'Bravo' objects>, '__doc__': None}`}

Атрибути класу Delta: {`'__module__': '__main__', '__doc__': 'Клас Delta наслідує клас Charlie'`}

Поля `__name__` і `__qualname__`

Клас Alpha, поле `__name__`: Alpha

Клас Alpha, поле `__qualname__`: Alpha

Клас Alpha.Bravo, поле `__name__`: Bravo

Клас Alpha.Bravo, поле `__qualname__`: Alpha.Bravo

Клас Delta, поле `__name__`: Delta

Клас Delta, поле `__qualname__`: Delta



Деякі поля (а саме, `__bases__` і `__mro__`) як значення повертають кортежі (набір елементів у круглих дужках). Якщо кортеж складається з одного елемента, за правилами синтаксису Python після цього елемента стоїть кома, хоча наступного елемента в кортежі немає.

Далі ми розглянемо спеціальні методи. Почнемо з методу `__call__()`, який призначено для виконання досить благородної місії. Якщо в класі описано цей метод, то екземпляр класу можна викликати як функцію. Невеликий приклад, який ілюструє, як визначається метод `__call__()`, наведено в лістингу 7.8.

Лістинг 7.8. Визначення методу `__call__()`

```
# Клас
class Box:
    # Конструктор
```

Phyton

```
def __init__(self, width, height, depth):
    # Поля екземпляра класу
    self.width=width
    self.height=height
    self.depth=depth
    # Визначення методу __call__()
def __call__(self):
    # Обчислюється добуток полів екземпляра
    volume=self.width*self.height*self.depth
    # Відображується результат обчислень
    print("Об'єм дорівнює",volume)
# Створюємо екземпляр класу
obj=Box(10,20,30)
# Викликаємо екземпляр класу як функцію
obj()
```

Результат виконання програмного коду такий:



Результат виконання програми (з лістингу 7.8)

Об'єм дорівнює 6000

У цьому прикладі ми експлуатуємо ідею з класом, який описує деякий «ящик» або «коробку» (в геометричному сенсі цього поняття). Клас називається `Box`. У класі описано конструктор, при виконанні якого полям `width`, `height` і `depth` присвоюються значення (аргументи конструктора). Для обчислення об'єму, замість того, щоб описувати метод екземпляра класу, визначаємо метод `__call__()`. Оскільки метод буде викликатися (неявно) через екземпляр класу, як аргумент методу зазначено посилання `self` на екземпляр класу. У тілі класу команда `volume=self.width*self.height*self.depth` обчислює добуток полів екземпляра класу (тобто об'єм «коробки») і записує в локальну змінну `volume`, після чого значення цієї змінної відображується у вікні виводу. На цьому опис класу `Box` закінчується.

Поза програмним кодом класу `Box` створюється екземпляр `obj` цього класу, а потім виконується команда `obj()` — тобто ми звертаємося до екземпляра класу так, як начеб це була функція. Оскільки в тілі класу `Box`

визначено метод `__call__()`, то автоматично викликається цей метод. У результаті у вікні виводу з'являється повідомлення з інформацією про об'єм «коробки».



Висловлюючись більш «технічно», команда `obj()` оброблюється, як команда `Box.__call__(obj)`.

Можна визначити метод `__call__()` так, щоб екземпляр класу не просто «викликався», а «викликався» з аргументами. Нескладно здогадати-ся, що для цього достатньо описати метод `__call__()` із додатковими (крім посилання на екземпляр класу) аргументами. Приклад такої ситуації наведено в лістингу 7.9.

Лістинг 7.9. Метод `call__()` із аргументами

```
# Клас
class Box:
    # Конструктор
    def __init__(self, width, height, depth):
        print("Об'єм дорівнює", self(width, height, depth))
    # Визначення методу __call__()
    def __call__(self, width, height, depth):
        # Поля екземпляра класу
        self.width=width
        self.height=height
        self.depth=depth
        # Обчислюємо добуток полів екземпляра
        volume=self.width*self.height*self.depth
        # Результат, що повертається
        return volume
# Створюємо екземпляр класу
obj=Box(10, 20, 30)
# Викликаємо екземпляр як функцію
print("Нове значення", obj(1, 2, 3))
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 7.9)

```
Об'єм дорівнює 6000
```

```
Нове значення 6
```

У цьому випадку ми так визначаємо метод `__call__()`, що під час виклику екземпляра класу (у форматі звертання до функції) необхідно передати три аргументи, які визначать значення полів екземпляра класу. Більше того, тепер ще й повертається результат — це об'єм (добуток полів екземпляра класу). Суттєво змінився і програмний код конструктора. Конкретніше, у тілі конструктора виконується всього одна команда `print("Об'єм дорівнює", self.width, height, depth)`, у якій є інструкція `self(width, height, depth)` виклику екземпляра класу `self` із аргументами `width`, `height` і `depth`. Таким чином, неявний виклик методу `__call__()` використовується вже в конструкторі — тобто при створенні екземпляра класу. Оскільки під час виклику екземпляра класу повертається результат, то у виразу `self(width, height, depth)` є числове значення — добуток полів екземпляра класу. Це значення буде відображатися у вікні виводу при створенні екземпляра класу.

Щодо безпосередньо програмного коду методу `__call__()`, то його описано з чотирма аргументами: аргумент `self` є посиланням на екземпляр класу, а аргументи `width`, `height` і `depth` задають значення полів екземпляра. У тілі методу командами `self.width=width`, `self.height=height` і `self.depth=depth` полям присвоюються значення, потім командою `volume=self.width*self.height*self.depth` обчислюється добуток полів, і, нарешті, інструкцією `return volume` це значення повертається як результат методу `__call__()`.

Тому, коли за допомогою команди `obj=Box(10, 20, 30)` створюється екземпляр класу, то під час виклику конструктора полям `width`, `height` і `depth` екземпляра присвоюються значення 10, 20 і 30 відповідно, обчислюється добуток цих полів, і відображується повідомлення. Під час виконання команди `print("Нове значення", obj(1, 2, 3))` зазначені поля отримують значення 1, 2 і 3, обчислюється нове значення для об'єму «коробки», і це значення відображується у вікні виводу.

Окрім методу `__call__()`, існують й інші корисні спеціальні методи, які умовно, залежно від їхнього, так би мовити, призначення, можна розбити

на групи. Так, є досить багато методів, призначених для зведення екземплярів класу до базових типів — таких, наприклад, як цілі числа або текст. У таблиці 7.2 наведено деякі з таких методів. Усі вони мають один аргумент — посилання на екземпляр класу (це, фактично, той екземпляр, який треба перетворити до базового типу). Явно ці методи зазвичай не викликаються. За яких обставин вони «вступають у гру», описано в таблиці.

Таблиця 7.2. Спеціальні методи зведення до типу

Метод	Опис
<code>__bool__()</code>	Метод для зведення екземпляра до логічного типу (тип <code>bool</code>). Викликається при використанні функції <code>bool()</code> або в інших випадках, коли виконується автоматичне зведення до логічного типу (наприклад, коли екземпляр указано в тому місці, де повинно бути логічне значення, — скажімо, в умовному операторі)
<code>__complex__()</code>	Метод для зведення до комплексного типу (тип <code>complex</code>). Метод викликається при використанні функції <code>complex()</code>
<code>__float__()</code>	Метод для зведення до числового типу <code>float</code> (формат числа з плаваючою крапкою). Метод викликається при використанні функції <code>float()</code>
<code>__int__()</code>	Метод для зведення до цілочислового типу (тип <code>int</code>). Метод викликається при використанні функції <code>int()</code>
<code>__str__()</code>	Метод для зведення екземпляра до текстового формату (тип даних <code>str</code>). Метод викликається при використанні функції <code>str()</code> або в тих випадках, коли виконується автоматичне зведення до текстового формату

Невеликий приклад з використанням цих методів наведено в лістингу 7.10. У цьому прикладі ми описуємо клас `MyClass`. Екземпляр, який створюється на основі цього класу, буде мати поле-спісок із числовими елементами. Далі ми встановлюємо такі «правила» зведення екземпляра до різних типів:

- Якщо екземпляр зводиться до ціличислового типу, то як значення повертається ціле число — кількість елементів у полі-списку.
- При зведенні екземпляра до типу `float` як значення повертається середнє арифметичне значення елементів у полі-списку.
- Якщо екземпляр зводиться до типу `complex`, то як значення повертається комплексне число, яке створюється за таким алгоритмом: дійсна частина комплексного числа — це елемент зі списку з максимальним значенням, а уявна частина комплексного числа — це елемент зі списку з мінімальним значенням.
- До логічного типу `bool` екземпляр буде зводитися так: якщо в полі-списку непарна кількість елементів, при зведенні екземпляра до типу `bool` повертається значення `True`, а якщо кількість елементів у списку парна, повертається значення `False`.
- Нарешті, при зведенні екземпляра до текстового типу буде повертатися текст, у якому, крім іншого, містяться значення елементів із поля-списку.

Тепер розглянемо програмний код.

Лістинг 7.10. Методи зведення до типу

```
# Клас
class MyClass:
    # Конструктор
    def __init__(self, nums):
        # Створюємо поле - пустий список
        self.nums=list()
        # Оператор циклу для перебору
        # елементів в аргументі nums
        # (це список або множина)
        for n in nums:
            # Додаємо новий елемент у список
            self.nums.append(n)
    # Метод для зведення до типу str
    def __str__(self):
        # Початкове значення текстового рядка
        txt="Значення поля-списку:\n| "

```

```

# Перебираємо елементи в списку nums - полі
# екземпляра класу
for n in self.nums:
    # Доповнюємо рядок новим текстом
    txt+=str(n)+" | "
# Результат методу
return txt

# Метод для зведення до типу int
def __int__(self):
    # Результат методу (кількість елементів
    # у списку nums - полі екземпляра класу)
    return len(self.nums)

# Метод для зведення до типу float
def __float__(self):
    # Середнє значення елементів у
    # списку nums - полі екземпляра
    avr=sum(self.nums)/int(self)
    # Результат методу
    return avr

# Метод для зведення до типу bool
def __bool__(self):
    # Якщо непарна кількість елементів
    if int(self)%2==1:
        # Результат методу
        return True
    # Якщо кількість елементів парна
    else:
        # Результат методу
        return False

# Метод для зведення до типу complex
def __complex__(self):
    # Мінімальне з чисел у списку
    mn=min(self.nums)
    # Максимальне з чисел у списку
    mx=max(self.nums)
    # Комплексне число
    z=complex(mx,mn)

```

Phyton

```
# Результат методу
return z

# Створюємо екземпляр класу
obj=MyClass({2.8,4.1,7.5,2.5,3.2})
# Виводимо на друк екземпляр
# (зведення до типу str)
print(obj)
# Зведення до типу int
print("Елементів у списку:",int(obj))
# Зведення до типу bool
if obj:
    print("Непарна кількість елементів")
# Зведення до типу float
print("Середнє значення:",float(obj))
# Зведення до типу complex
print("Максимум і мінімум:",complex(obj))
```

У результаті виконання цього коду отримуємо таке:



Результат виконання програми (з лістингу 7.10)

Значення поля-списку:

| 2.5 | 7.5 | 2.8 | 4.1 | 3.2 |

Елементів у списку: 5

Непарна кількість елементів

Середнє значення: 4.02

Максимум і мінімум: (7.5+2.5j)

У класу MyClass є конструктор. У конструктора є аргумент nums. Цей аргумент позначає деякий набір елементів для формування поля-списку. Виходитимемо з того, що це список або множина. Але під час створення екземпляра класу елементи об'єднуємо у вигляді списку. Для цього в тілі конструктора за допомогою команди `self.nums=list()` в екземпляра `self` створюємо поле `nums`, яке спочатку є порожнім списком. Потім запускається оператор циклу, в якому змінна `n` перебирає значення зі списку або множини `nums`, що є аргументом конструктора. За кожний цикл командою `self.nums.append(n)` у поле-список `nums` екземпляра `self` додається за допомогою методу `append()` новий елемент (zmінна `n`).

Метод `__str__()` для перетворення на тип `str` також містить оператор циклу. У тілі методу спочатку за допомогою команди `txt="Значення поля-списку:\n| "` визначається початкове текстове значення для змінної `txt`, а потім в операторі циклу, в якому змінна `n` перебирає значення з поля-списку `nums` екземпляра `self`, командою `txt+=str(n)+" | "` додається текст зі значенням елемента і роздільником (у вигляді вертикальної риски). Після закінчення оператора циклу змінна `txt` командою `return txt` повертається як результат методу.

Щоб мати можливість виконувати перетворення екземпляра на тип `int` у класі `MyClass` описується метод `__int__()`. У тілі методу всього одна команда `return len(self.nums)`, яка результатом методу повертає кількість елементів у списку `nums` із екземпляра `self`. Тут для обчислення кількості елементів використовується вбудована функція `len()`.

Метод `__float__()` призначений для зведення до типу `float`. У тілі методу для значень елементів поля-списку `nums` екземпляра `self` обчислюється середнє арифметичне: суму елементів обчислюємо за допомогою вбудованої функції `sum()`, і отримане значення ділимо на кількість елементів у списку. Причому для визначення кількості елементів ми використовуємо інструкцію `int(self)` явного зведення екземпляра `self` до цілочислового типу (нагадаємо, що ця операція перевизначена нами так, що її результатом є кількість елементів у полі-списку екземпляра). Результат записується в змінну `avr`, яка й повертається як результат методу.

У тілі методу `__bool__()`, який описується, щоб зводити екземпляр до типу `bool`, запускається умовний оператор, у якому перевіряється умова `int(self)%2==1`: остача від ділення на 2 кількості елементів у полі-списку екземпляра `self` повинна дорівнювати 1. Таке трапляється, якщо кількість елементів непарна. Як і раніше, для визначення кількості елементів ми використовуємо явне зведення екземпляра `self` до типу `int` за допомогою функції `int()`. Незайвим буде підкреслити, що це можливо, оскільки в тілі класу описано метод `__int__()`.

Так ось, якщо кількість елементів непарна, в умовному операторі інструкція `return True` для методу `__bool__()` повертає значення `True`. В іншому випадку, повертається значення `False`.

Метод для перетворення на тип `complex` називається `__complex__()`. У тілі методу командою `mn=min(self.nums)` визначається мінімальне значення з набору елементів у полі-списку `nums` екземпляра `self`, а командою `mx=max(self.nums)` визначається максимальне значення з набору елементів у полі-списку `nums` екземпляра `self`. Потім ми створюємо комплексне число за допомогою команди `z=complex(mx,mn)`. Це комплексне число повертається як результат методу `__complex__()`.



Ми описуємо метод `__complex__()`, щоб можна було використовувати функцію `complex()`, передаючи їй аргументом посилання на екземпляр класу. В описі методу `__complex__()` ми викликаємо функцію `complex()`, але тут проблеми немає, оскільки ми цю функцію викликаємо у «стандартній», «штатній» ситуації, з двома аргументами типу `float`.

Екземпляр класу створюється командою `obj=MyClass({2.8, 4.1, 7.5, 2.5, 3.2})`. Як аргумент конструктору передається множина з числових значень, хоча це міг би бути й список. Що б не було, екземпляр `obj` створено, у цього екземпляра є поле-список `nums`, і ще цей екземпляр можна зводити до базових типів. Саме ці можливості перевіряємо. Для початку намагаємося «надрукувати» екземпляр, для чого використовуємо команду `print(obj)`. Хоча формально ми тут функцію `str()` із аргументом-екземпляром не викликаємо, за контекстом команди `print(obj)` запускається автоматичне зведення до типу `str`. Тому під час виконання команди `print(obj)` у вікні виводу фактично відображується текстове значення, яке повертається як результат методом `__str__()` із аргументом — екземпляром `obj`. Аналогічна ситуація з автоматичним зведенням (тільки тепер до типу `bool`) має місце під час виконання умовного оператора, в якому як умову вказано екземпляр `obj`. У всіх інших випадках зведення виконується явно з викликом відповідної функції і передачею посилання на екземпляр класу як аргументу функції.

У розглянутому вище прикладі для визначення кількості елементів у списку-полі екземпляра класу ми визначали метод `__int__()`, у якому викликалася функція `len()`, яка, у свою чергу, визначала кількість елементів у полі-списку екземпляра класу. Можна було піти іншим шляхом: описати в тілі класу метод `__len__()`. Цей метод викликається, якщо викликається функція `len()` із аргументом — екземпляром відповідного класу. Іншими словами, якщо описати в класі метод

`__len__()`, то можна буде викликати функцію `len()` для екземпляра цього класу. У лістингу 7.11 наведено приклад опису методу `__len__()`. Крім цього методу, також описано методи (із тих, що поки нам незнайомі) `__index__()` (викликається при використанні функцій `bin()`, `oct()` і `hex()`) і `__round__()` (викликається при використанні функції `round()`).

Лістинг 7.11. «Округлення» екземпляра й інші операції

```
# Клас
class MyClass:
    # Конструктор
    def __init__(self,txt):
        # Присвоювання значення полю
        # екземпляра класу
        self.name=txt
    # Метод для зведення до текстового типу
    def __str__(self):
        # Результат методу - значення
        # поля name екземпляра класу
        return self.name
    # Метод для обчислення "довжини"
    # екземпляра класу функцією len()
    def __len__(self):
        # Результат методу - кількість
        # символів у полі name екземпляра класу
        return len(self.name)
    # Метод, який викликається при використанні
    # функцій bin(), oct() і hex()
    def __index__(self):
        # Кількість пробілів плюс одиниця
        p=self.name.count(" ") +1
        # Результат методу
        return p
    def __round__(self):
        # Присвоювання значення полю name
        self.name="Скидання значення"
```

Phyton

```
# Результат - посилання на екземпляр класу
return self

# Початкове текстове значення
txt="Раз, два, три, чотири, п'ять. "
# Уточнюємо текстове значення
txt+="\nП'ятеро пташат летять."
# Створюємо екземпляр класу
obj=MyClass(txt)
# Екземпляр "друкується" у вікні виводу
print(obj)
# Обчислюємо кількість символів у полі name
print("Кількість букв (символів):",len(obj))
# Кількість слів (пробілів) у полі name
print("Кількість слів:",obj.__index__())
# Двійковий код
print("У двійковому коді:",bin(obj))
# Вісімковий код
print("У вісімковому коді:",oct(obj))
# Шістнадцятковий код
print("У шістнадцятковому коді:",hex(obj))
# Виконуємо "округлення" екземпляра класу
print(round(obj))
# Виводимо "на друк" екземпляр класу
print(obj)
```

Результат виконання програми такий:



Результат виконання програми (з лістингу 7.11)

```
Раз, два, три, чотири, п'ять.
П'ятеро пташат летять.
Кількість букв (символів): 53
Кількість слів: 8
У двійковому коді: 0b1000
У вісімковому коді: 0o10
У шістнадцятковому коді: 0x8
Скидання значення
Скидання значення
```

Клас називається `MyClass`, і в цьому класі є конструктор. Конструктору передається текстовий аргумент, посилання на який присвоюється як значення полю `name` екземпляра класу. Також заради зручності в класі описано метод `__str__()`, завдяки чому функції `print()` можна передавати аргументом екземпляр класу. При цьому буде відображуватися значення, на яке посилається поле `name` екземпляра класу.

Метод `__len__()` для обчислення «довжини» екземпляра класу (якщо екземпляр передається як аргумент функції `len()`) містить усього одну інструкцію `return len(self.name)`: як результат методу повертається довжина (кількість символів) текстового поля `name` екземпляра `self`. Тому кожного разу, коли ми викликаємо функцію `len()` і передаємо їй як аргумент посилання на екземпляр класу, результатом буде кількість символів у полі `name` цього екземпляра.

Метод `__index__()`, який викликається при використанні функцій `bin()`, `oct()` і `hex()`, містить команду `p=self.name.count(" ") + 1`, якою в змінну `p` записується кількість пробілів у поле `name` екземпляра `self`, плюс одиниця. Кількість пробілів підраховується за допомогою вбудованого методу `count()`. Значення змінної `p` повертається як результат методу.



Якщо виходити з того, що між словами в тексті знаходиться пробіл, то кількість пробілів на одиницю менша, ніж кількість слів у тексті. У цьому сенсі результат, який повертається методом `__init__()`, можна інтерпретувати (з певною умовністю, звичайно) як кількість слів у текстовому полі `name` екземпляра.

У тілі методу `__round__()` команда `self.name="Скидання значення"` полю `name` присвоює значення, і потім команда `return self` посилання на екземпляр `self`, із якого викликається метод, повертає як результат методу.



Як зазначалося раніше, метод `__round__()` зали чається, коли функція `round()` викликається з екземпляром класу (у цьому разі `MyClass`). Наприклад, якщо посилання на екземпляр позначити `obj`, то інструкція

`round(obj)` насправді означає виконання команди `obj.__round__()`. При виконанні цієї команди (через те, як ми описали метод `__self__()`) полю `name` присвоюється значення "Скидання значення", а результатом виразу `round(obj)` є посилання на екземпляр `obj`.

Для перевірки функціональності описаних методів у змінну `txt` записується текстове значення, а потім на основі цього текстового значення за допомогою команди `obj=MyClass(txt)` створюється екземпляр `obj` класу `MyClass`. Далі йде серія команд, у яких використовується посилання на цей екземпляр. Так, спочатку виконується команда `print(obj)`. У цьому випадку «у гру» вступає метод `__str__()`. В результаті відображується значення поля `name` екземпляра `obj`. Кількість символів у полі `name` відображується при виконанні команди `print("Кількість букв (символів):", len(obj))`. У цій команді використано інструкцію `len(obj)`, яка обчислюється шляхом виклику методу `__len__()` із екземпляра `obj`. Кількість слів відображаємо за допомогою команди `print("Кількість слів:", obj.__index__())` — тут явно викликаємо метод `__index__()`. При обчисленні інструкцій `bin(obj)`, `oct(obj)` і `hex(obj)` теж викликається цей метод, але числовий результат (а це, з математичної точки зору, кожний раз один і той же результат) відображується відповідно у двійковій, вісімковій і шістнадцятковій системах числення.

Нарешті, при виконанні команди `print(round(obj))` полю `name` екземпляра `obj` присвоюється значення "Скидання значення". Оскільки результатом `round(obj)` є посилання на екземпляр `obj`, а також завдяки описаному в класі `MyClass` методу `__str__()`, під час виконання команди `print(round(obj))` з'являється повідомлення Скидання значення. Аби пересвідчитися, що екземпляр `obj` змінився, виконуємо команду `print(obj)`.

Група методів `__setitem__()`, `__getitem__()` і `__delitem__()` автоматично викликається відповідно під час присвоювання значення екземпляру через індекс, зчитування значення екземпляра через індекс і видалення значення екземпляра через індекс.



Ці ж методи використовують для обробки операцій присвоювання значення, зчитування значення і видалення значення за ключем.

У лістингу 7.12 наведено приклад використання цих методів. Там ми описуємо клас MyClass для створення екземплярів з полем nums, що є числовим списком. Кількість елементів у списку визначається полем класу Nmax. Описуючи методи `__setitem__()`, `__getitem__()` і `__delitem__()`, ми хочемо домогтися ефекту, щоб можна було звертатися до елементів поля nums екземпляра (для присвоювання значення, зчитування значення і видалення значення), вказуючи індекс безпосередньо для екземпляра класу. Причому якщо індекс виходить за допустимі межі, виконується циклічна перестановка. Програмний код виглядає так:

Лістинг 7.12. Операції з екземплярами через індекс

```
# Клас
class MyClass:
    # Поле класу
    Nmax=5
    # Конструктор
    def __init__(self):
        # Кількість елементів
        n=MyClass.Nmax
        # Список із нульовими елементами
        self.nums=[0 for i in range(n)]
    # Метод зведення до текстового типу
    def __str__(self):
        # Текстова змінна
        txt="|"
        # Формуємо текст
        for s in self.nums:
            # Додаємо до тексту фрагмент
            txt+=" "+str(s)+" |"
        # Результат методу
        return txt
    # Метод для присвоювання значення за індексом
    def __setitem__(self,i,v):
```

Phyton

```
# Остача від ділення
k=i % len(self.nums)
# Присвоювання значення
self.nums [k]=v

# Метод для зчитування значення за індексом
def __getitem__(self,i):
    # Остача від ділення
    k=i % len(self.nums)
    # Значення елемента
    return self.nums[k]

# Метод для видалення значення за індексом
def __delitem__(self,i):
    # Остача від ділення
    k=i % len(self.nums)
    # Нове значення елемента
    self.nums [k] = "*""

# Створюється екземпляр класу
obj=MyClass()
# Вміст списку
print(obj)
# Нові значення елементів списку
obj[0]=100
obj[2]=-3
obj[24]=123
# Вміст списку
print(obj)
# Зчитування значень елементів списку
print("Елемент із індексом 4:",obj[4])
print("Елемент із індексом 7:",obj[7])
# Видалення елементів списку
del obj[0]
del obj[9]
# Вміст списку
print(obj)
```

Результат виконання цього коду наведено нижче:

■ Результат виконання програми (з листингу 7.12)

```
| 0 | 0 | 0 | 0 | 0 |
| 100 | 0 | -3 | 0 | 123 |
Елемент із індексом 4: 123
Елемент із індексом 7: -3
| * | 0 | -3 | 0 | * |
```

У класі MyClass полю `Nmax` присвоюємо значення 5. Це означає, що по-ле-спісок екземпляра класу буде містити 5 елементів. Під час створення екземпляра всі елементи цього списку отримують нульові значення. Зокрема, у тілі конструктора створюється список і заповнюється нулями (усе це робиться за допомогою генератора списків). Метод `__str__()` описано так, щоб під час передачі функції `print()` посилання на екземпляр класу у вікні виводу відображався вміст списку `nums` — поля екземпляра класу.

Усе, що реалізовано в тілі методу `__setitem__()`, — це, насправді, те, що відбувається під час спроби виконати команду вигляду `екземпляр[індекс]=значення`. Усі три параметри (посилання на екземпляр класу, індекс і значення, яке присвоюється) описано як аргументи методу `__setitem__()`, — відповідно `self`, `i` і `v`. У тілі методу командою `k=i % len(self.nums)` обчислюється остача від ділення індексу `i` на значення `len(self.nums)` (кількість елементів у списку `nums`). Потім командою `self.nums[k]=v` елементу в цьому списку з індексом `k` присвоюється значення `v`.

Дещо схоже відбувається під час виконання програмного коду методу `__getitem__()`. У цьому разі йдеться про обчислення значення виразу вигляду `екземпляр[індекс]`. Відмінність у порівнянні з попереднім випадком у тому, що раніше цьому виразу присвоювалося значення, а тепер для цього виразу значення обчислюється (читування значення). Тому в методу `__getitem__()` два аргументи: посилання на екземпляр класу `self` і значення індексу `i`. У тілі методу виконуються всього дві команди: командою `k=i % len(self.nums)` обчислюється індекс `k` із урахуванням циклічної перестановки, і командою `return self.nums[k]` повертається результат методу — значення елемента з індексом `k` зі списку `nums` екземпляра `self`.

Метод `__delitem__()` автоматично викликається при спробі видалення значення за допомогою команди вигляду `del` екземпляр[індекс]. У методу (під час опису) два аргументи: посилання на екземпляр класу й індекс (елемента, який видаляється). У тілі методу традиційно команда `k=i % len(self.nums)` «уточнює» індекс, а потім виконується присвоювання `self.nums[k] = "*"`. Таким чином, якщо буде здійснено спробу видалити елемент, то насправді цьому елементу як значення буде присвоєно текст `"*"`.



Коли ми говоримо про видалення елементів, то фактично йдеться про команду вигляду `del` екземпляр[індекс]. Якщо ми видаляти мемо елементи, напримки звертаючись до поля-списку екземпляра, видалення буде виконуватися так, як і повинно було б бути. Це ж зауваження стосується й присвоювання значень через індекс (команда `екземпляр[індекс]=значення`), і зчитування значення через індекс (команда `екземпляр[індекс]`). Крім того, зовсім необов'язково, щоб в екземпляра було поле-спісок. У принципі, звертання до екземпляра через індекс можна реалізувати по-іншому, без залучення вбудованих в екземпляр списків.

Після того, як команда `obj=MyClass()` створить екземпляр `obj` класу `MyClass`, усі елементи (їх усього 5) списку `nums` екземпляра будуть нульовими. Значення елементів списку `nums` відображуються при виконанні команди `print(obj)`. Потім декілька команд виконують присвоювання значень елементам списку `nums`, причому звертання до елементів відбувається через зазначення індексу для екземпляра класу. Так, команда `obj[0]=100` елементу `nums[0]` (поле-спісок `nums` екземпляра `obj`) присвоює значення 100, команда `obj[2]=-3` присвоює значення -3 елементу `nums[2]`, і, зрештою, команда `obj[24]=123` елементу `nums[4]` присвоює значення 123.



Щодо команди `obj[24]=123`: остача від ділення 24 на 5 є 4. Тому йдеться про присвоювання значення елементу масиву `nums` із індексом 4.

Після внесення змін перевіряємо вміст списку `nums` за допомогою команди `print(obj)`. Можна зчитувати значення й поелементно, звертаючись

«персонально» до того чи іншого елемента, — як це робиться в інструкціях `obj[4]` і `obj[7]` (елементи `nums[4]` і `nums[2]` відповідно). При спробі видалення елементів списку командами `del obj[0]` і `del obj[9]` елементи `nums[0]` і `nums[4]` отримують значення `"*"`. Результат перевіряємо за допомогою команди `print(obj)`.



Інструкції `obj[7]` і `obj[9]` означають звертання до елементів `nums[2]` і `nums[4]` відповідно: остача від ділення 7 на 5 дорівнює 2, а остача від ділення 9 на 5 дорівнює 4.

Завдяки опису спеціальних методів `__getattr__()`, `__getattribute__()`, `__setattr__()` і `__delattr__()` можна суттєво урізноманітнити операції звертання до атрибутів екземпляра класу. Метод `__setattr__()` викликається під час присвоювання значення атрибуту екземпляра класу. Метод `__delattr__()` викликається під час видалення атрибута екземпляра класу. Під час звертання (для зчитування значення) до неіснуючого атрибута викликається метод `__getattr__()`. Нарешті, під час звертання до будь-якого атрибута викликається метод `__getattribute__()`.



Метод `__getattr__()` викликається під час звертання до неіснуючого атрибута екземпляра класу, якщо в класі не описано метода `__getattribute__()`. Якщо метод `__getattribute__()` у класі описано, то викликається він, причому під час звертання до будь-якого атрибута — як існуючого, так і неіснуючого. Метод `__getattr__()`, навіть якщо його описано в класі, при цьому ігнорується.

Із методом `__getattribute__()` пов'язана певна проблема: якщо в тілі цього методу виконується звертання до атрибута екземпляра класу з використанням «крапкового синтаксису» (тобто у форматі `екземпляр.атрибут`), то виходить нескінчений циклічний виклик методу `__getattribute__()`. Дійсно, якщо викликається метод `__getattribute__()`, а під час виконання цього методу виконується звертання до атрибута екземпляра класу, то метод буде викликано знову. Тобто в процесі виконання метод фактично викликає сам себе. Але на цьому процес не закінчується. Викликаний метод знову викликає цей самий метод і так до нескінченості.

На перший погляд, ситуація видається безнадійною, але це, зрозуміло, не так. Просто звертання до атрибутів екземпляра класу слід виконувати, явно викликаючи метод `__getattribute__()` як функцію з класу `object` із явним зазначенням посилання на екземпляр класу і його атрибут. Оскільки йдеться про звертання до атрибута в тілі методу `__getattribute__()`, то посилання на екземпляр класу буде називатися `self`. Якщо йдеться про звертання до атрибута цього екземпляра, то звертання до цього атрибута буде виглядати як `object.__getattribute__(self, атрибут)`.

Дещо схоже відбувається, коли в тілі методу `__setattr__()` (який «відповідає» за присвоювання значень атрибутам екземплярів) виконується присвоювання значення атрибуту з використанням «крапкового синтаксису». Проблема така: під час виконання методу `__setattr__()`, якщо присвоюється значення атрибуту, автоматично викликається метод `__setattr__()`, у тілі якого викликається метод `__setattr__()`, і так далі. Щоб цього не відбувалося, під час присвоювання значення атрибуту екземпляра класу в тілі методу `__setattr__()` слід «добиратися» до потрібного атрибута через спеціальне поле `__dict__`, значенням якого є словник із назв атрибутів екземпляра та їхніх значень. Якщо йдеться про присвоювання значення атрибуту, то в тілі методу `__setattr__()` відповідна команда може виглядати як `self.__dict__[атрибут]=значення` (тут через `self` позначено посилання на екземпляр класу). Формально в цьому випадку ми в словнику `__dict__` змінюємо значення елемента з ключем атрибут. Але оскільки набір атрибутів і значень атрибутів екземпляра реалізується саме через цей словник, то потрібного ефекту буде досягнуто — атрибут екземпляра змінює значення.

Не варто видаляти поля за допомогою оператора `del` у тілі методу `__delattr__()`. У цьому разі метод `__delattr__()` буде викликати сам себе. Якщо треба видалити поле в тілі методу `__delattr__()`, краще зробити це, видаливши відповідний елемент зі словника `__dict__`.

Невеликий приклад із описом методів `__getattribute__()`, `__setattr__()` і `__delattr__()` наведено в лістингу 7.13. У цьому прикладі створюється клас `MyClass`. У класі описано конструктор — такий, що під час створення екземпляра класу полю `name` екземпляра присвоюється значення. Також у класі описано метод `__str__()`, завдяки чому при передачі екземпляра аргументом методу `print()` відображується значення поля `name` цього екземпляра.

Ми також хочемо досягти наступної реакції на дії з атрибутами екземпляра класу:

- Значення поля `name` можна змінювати.
- При спробі створити (шляхом присвоювання значення) нове поле з'являється повідомлення Операцію не дозволено!, і нове поле не створюється.
- Якщо під час зчитування значення поля виконується звертання до неіснуючого поля, з'являється повідомлення Такого поля немає!.
- При спробі видалити поле екземпляра з'являється повідомлення Видаляти поля заборонено!.

Власне, для розв'язання цих задач ми їй описуємо в класі `MyClass` методи `__getattr__()`, `__setattr__()` і `__delattr__()`. Як саме їх описано, показано нижче:

Лістинг 7.13. Звертання до полів екземпляра

```
# Клас
class MyClass:
    # Конструктор
    def __init__(self, name):
        # Полю екземпляра
        # присвоюється значення
        self.name=name

    # Метод для зведення екземпляра
    # до текстового значення
    def __str__(self):
        # Результат методу
        return self.name

    # Метод для обробки ситуації, коли
    # атрибуту присвоюється значення
    def __setattr__(self, attr, val):
        # Якщо значення присвоюється
        # полю name
        if attr=="name":
            self.__dict__[attr]=val
        # Якщо значення присвоюється не полю name
        else:
            print("Операцію не дозволено!")
```

Phyton

```
# Метод для обробки ситуації, коли
# зчитується значення атрибута
def __getattr__(self,attr):
    # Результат методу
    return "Такого поля немає!"
# Метод для обробки ситуації, коли
# атрибут видаляється
def __delattr__(self,attr):
    # Відображується повідомлення
    print("Видаляти поля заборонено!")
# Створюється екземпляр класу
obj=MyClass("Вихідне значення")
# Перевіряємо значення поля name
print(obj)
# Нове значення поля name
obj.name="Нове значення"
# Перевіряємо значення поля name
print(obj)
# Присвоюємо значення полю number
obj.number=100
# Перевіряємо значення поля number
print(obj.number)
# Видаляємо поле name
del obj.name
# Перевіряємо значення поля name
print(obj)
```

Результат виконання програмного коду такий:

■ Результат виконання програми (з лістингу 7.13)

```
Вихідне значення
Нове значення
Операцію не дозволено!
Такого поля немає!
Видаляти поля заборонено!
Нове значення
```

Розглянемо програмні коди методів `__getattr__()`, `__setattr__()` і `__delattr__()`. Найпростіший код у методів `__getattr__()` і `__delattr__()`: у тілі методу `__getattr__()` результат повертає команда `return "Такого поля немає!"`, а в тілі методу `__delattr__()` виконується єдина команда `print("Видаляти поля заборонено!")`. Із методом `__setattr__()` усе небагато складніше: в умовному операторі перевіряється умова `attr=="name"`, яка істинна, якщо запитуваний атрибут (аргумент методу `attr`) — це поле `name`. У такому разі виконується команда `self.__dict__[attr]=val`. Тут у словнику `__dict__` для екземпляра `self` елементу з ключем `attr` присвоюється значення `val` (аргумент методу `__setattr__()`). Якщо умова `attr=="name"` хибна (тобто атрибут `attr` не є полем `name`), виконується команда `print("Операцію не дозволено!")`.

Після опису класу `MyClass` командою `obj=MyClass("Вихідне значення")` створюється екземпляр `obj`, поле `name` якого отримує значення `"Вихідне значення"`. Перевірити значення поля `name` екземпляра `obj` можна за допомогою команди `print(obj)`.

Командою `obj.name="Нове значення"` полю `name` присвоюється нове значення. Ця операція проходить успішно. Але при спробі виконати команду `obj.number=100` поле `number` в екземпляра `obj` не створюється, і, зрозуміло, значення йому не присвоюється (замість цього з'являється повідомлення Операцію не дозволено!). При спробі «прочитати» значення неіснуючого поля `number` командою `print(obj.number)` з'являється повідомлення Такого поля немає!. Якщо спробувати видалити поле `name` в екземпляра `obj` за допомогою команди `del obj.name`, поле `name` не видаляється, і з'являється повідомлення Видаляти поля заборонено!.

Ще один спосіб обробки операцій присвоювання значень полям, зчитування значень полів і видалення полів, наведено в наступному прикладі. Як і в попередньому випадку, ми описуємо методи `__setattr__()` і `__delattr__()`, а замість методу `__getattr__()` описуємо метод `__getattribute__()`.



Нагадаємо, що метод `__getattr__()` викликається при зчитуванні значення неіснуючого поля, у той час як метод `__getattribute__()` викликається при зчитуванні значення будь-якого поля — як існуючого, так і не існуючого.

Отже, у класі `MyClass` описуються методи `__setattr__()`, `__getattribute__()` і `__delattr__()`.

Під час виконання методу `__setattr__()`:

- з'являється повідомлення про початок виконання методу;
- з'являється повідомлення про те, якому полю яке значення присвоюється;
- виконується присвоювання значення полю;
- з'являється повідомлення про закінчення виконання методу.

Під час виконання методу `__getattribute__()`:

- з'являється повідомлення про початок виконання методу;
- з'являється повідомлення про те, значення якого поля зчитується;
- виконується спроба прочитати значення поля;
- якщо поле існує, з'являється повідомлення про закінчення виконання методу і повертається значення поля;
- якщо поля не існує, з'являється повідомлення про закінчення виконання методу і повертається текстове значення з інформацією про те, що поля не існує.

Під час виконання методу `__delattr__()`:

- з'являється повідомлення про початок виконання методу;
- з'являється повідомлення про те, яке поле видаляється;
- виконується спроба видалити поле;
- якщо поле існує і його видалення проходить успішно, з'являється повідомлення про закінчення виконання методу;
- якщо поле не існує, з'являється повідомлення про неможливість видалити поле, а потім повідомлення про закінчення виконання методу.

Увесь програмний код наведений в лістингу 7.14 і виглядає так:

Лістинг 7.14. Зчитування значення поля

```
# Клас
class MyClass:
    # Метод викликається, якщо полю
    # присвоюється значення
    def __setattr__(self, attr, val):
        print("Виконується метод __setattr__():")
        txt="*\tПолю "+str(attr)
        txt+=" присвоюється значення "+str(val)
        print(txt)
        # Присвоювання значення полю
        self.__dict__[attr]=val
        print("Метод __setattr__() виконано.")

    # Метод викликається, якщо
    # зчитується значення поля
    def __getattribute__(self, attr):
        print("Виконується метод __getattribute__():")
        txt="*\tЗчитується значення поля "+str(attr)
        print(txt)
        # Результат методу
        try:
            # Значення поля - якщо поле існує
            res=object.__getattribute__(self, attr)
        except AttributeError:
            # Якщо поле не існує
            res="У екземпляра поля "+str(attr)+" немає!"
        print("Метод __getattribute__() закінчує роботу.")
        # Результат методу
        return res

    # Метод викликається, якщо
    # поле видаляється
    def __delattr__(self, attr):
        print("Виконується метод __delattr__():")
        txt="*\tВидаляється поле "+str(attr)
        print(txt)
        try:
            # Видалення поля - якщо поле існує
            del self.__dict__[attr]
```

Phyton

```
except KeyError:  
    # Якщо такого поля не існує  
    print("Не можна видалити поле "+str(attr))  
    print("Метод __delattr__() виконано.")  
  
# Створюється екземпляр класу  
obj=MyClass()  
  
# Полю name присвоюється значення  
obj.name="Python"  
  
# Перевіряється значення поля name  
print("Значення поля name:", obj.name)  
  
# Видаляється поле name  
del obj.name  
  
# Перевіряється значення поля name  
print(obj.name)  
  
# Повторно видаляється поле name  
del obj.name
```

Результат виконання програмного коду наведено нижче:

■ Результат виконання програми (з листингу 7.14)

```
Виконується метод __setattr__():  
* Полю name присвоюється значення Python  
Виконується метод __getattribute__():  
* Зчитується значення поля __dict__  
Метод __getattribute__() закінчує роботу.  
Метод __setattr__() виконано.  
Виконується метод __getattribute__():  
* Зчитується значення поля name  
Метод __getattribute__() закінчує роботу.  
Значення поля name: Python  
Виконується метод __delattr__():  
* Видаляється поле name  
Виконується метод __getattribute__():  
* Зчитується значення поля __dict__  
Метод __getattribute__() закінчує роботу.  
Метод __delattr__() виконано.  
Виконується метод __getattribute__():
```

* Зчитується значення поля name

Метод `__getattribute__()` закінчує роботу.

В екземпляра поля name немає!

Виконується метод `__delattr__()`:

* Видаляється поле name

Виконується метод `__getattribute__()`:

* Зчитується значення поля `__dict__`

Метод `__getattribute__()` закінчує роботу.

Не можна видалити поле name

Метод `__delattr__()` виконано.

Щоб зrozуміти, чому з'являються саме такі повідомлення, до уваги треба взяти такі обставини.

- Після створення екземпляра класу командою `obj=MyClass()` у цього екземпляра немає полів (тих, що створені користувачем). Під час виконання команди `obj.name="Python"` в екземпляра `obj` з'являється поле name зі значенням "Python". При цьому викликається метод `__setattr__()`, у тілі якого виконується звертання до спеціального поля `__dict__`. Звертання до поля `__dict__`, у свою чергу, приводить до виклику методу `__getattribute__()`.
- Під час перевірки значення поля name командою `print("Значення поля name:", obj.name)` викликається метод `__getattribute__()`.
- Під час видалення поля name командою `del obj.name` викликається метод `__delattr__()`. Оскільки у процесі видалення поля виконується звернення до спеціального поля `__dict__`, то автоматично викликається і метод `__getattribute__()`.
- Після видалення поля name значення цього поля перевіряється за допомогою команди `print(obj.name)`. Як наслідок, викликається метод `__getattribute__()`.
- Спроба повторно виділити поле name за допомогою команди `del obj.name` приводить до виклику методу `__delattr__()`, а під час виконання цього методу — до виклику методу `__getattribute__()`.

Далі ми познайомимося з одним дуже цікавим механізмом, який доступний у процесі роботи з класами й екземплярами класів у Python. Йтиметься про *перевантаження операторів*.

Перевантаження операторів

Давайте спершу перекусимо, а дружелюбність проявите потім.

З телесеріалу «Альф»

У Python існує можливість визначати для екземплярів класів такі операції, як додавання, множення, віднімання, ділення та ряд інших. Тобто ми маємо змогу так описати клас, що екземпляри цього класу можна буде додавати, віднімати, множити, ділити і так далі — як, наприклад, звичайні числа. Що виходить у результаті — це інше питання, і відповідь на нього цілком залежить від нас і нашої фантазії. Щоб та чи інша операція (наприклад, додавання) стала доступною для виконання з екземплярами класу, достатньо в класі описати певний спеціальний метод (кожному оператору відповідає метод, який автоматично викликається під час обробки виразу з цим оператором). Якщо такий метод у класі описано, то операція, яка відповідає цьому методу, буде доступна для екземплярів класу. Наприклад, оператору додавання + відповідає метод `__add__()`. Якщо в класі описано метод `__add__()`, то під час розрахунку виразу, в якому до екземпляра класу щось додається, буде викликано (автоматично) метод `__add__()`. Невеликий приклад із класом, у якому описано метод `__add__()`, наведено в лістингу 7.15.



Лістинг 7.15. Перевантаження оператора додавання

```
# Клас із описом методу __add__()
class Adder:
    # Конструктор
    def __init__(self, number):
        # Полю екземпляра присвоюється значення
        self.number=number
```

```

# Метод для зведення до текстового
# типу
def __str__(self):
    # Формується текст
    txt="Значення поля number = "
    txt+=str(self.number)
    # Результат методу
    return txt

# Опис методу для операції додавання
def __add__(self,x):
    # Обчислюється числове значення
    number=self.number+x
    # Створюється екземпляр класу
    tmp=Adder(number)
    # Результат методу - посилання на
    # екземпляр класу
    return tmp

# Створюється екземпляр класу
a=Adder(10)
# До екземпляра класу додається число
b=a+5
# Перевіряємо поле number 1-го екземпляра
print(a)
# Перевіряємо поле number 2-го екземпляра
print(b)

```

Результат виконання програмного коду такий:

■ Результат виконання програми (з листингу 7.15)

```

Значення поля number = 10
Значення поля number = 15

```

У класі Adder є конструктор, під час виконання якого полю number екземпляра присвоюється значення. Метод `__str__()` описано так, що при передачі екземпляра класу аргументом методу `print()` відображується значення поля `number` екземпляра.

Цікавим є метод `__add__()`, описаний із двома аргументами: аргумент `self` традиційно позначає посилання на екземпляр класу, з якого викликається метод, а аргумент `x` позначає числове (як ми припускаємо) значення, яке додається до екземпляра класу.

У тілі методу командою `number=self.number+x` обчислюється сума значення поля `number` екземпляра `self` і значення аргументу `x`. Результат записується в локальну змінну `number`. Це значення передається аргументом конструктору під час створення нового екземпляра класу за допомогою команди `tmp=Adder(number)`. Цей екземпляр класу повертається результатом методу командою `return tmp`.

Таким чином, при додаванні до екземпляра класу числового значення створюється новий екземпляр класу, поле `number` якого дорівнює сумі значень поля `number` вихідного екземпляра і числа, яке додається до цього екземпляра.

Поза тілом класу командою `a=Adder(10)` створюємо екземпляр `a` класу `Adder` зі значенням 10 для поля `number`. Завдяки тому, що в класі `Adder` описано метод `__add__()`, стає можливою команда `b=a+5`, якою обчислюється «сума» екземпляра `a` і числа 5, а результат — посилання на новий екземпляр класу — записується у змінну `b`. Після цього змінні `a` і `b` посилаються на екземпляри класу `Adder`. Перевірити, що йдеться про різні екземпляри, легко: достатньо виконати команди `print(a)` і `print(b)`.



Якщо ми замість того, щоб до екземпляра класу додавати число, спробуємо до числа додати екземпляр класу (наприклад, спробуємо виконати команду `b=5+a`), з'явиться повідомлення про помилку. Річ у тім, що метод `__add__()` «оброблює» тільки додавання до екземпляра числа, але не на впаки. Щоб можна було до числа додавати екземпляр, треба в класі `Adder` описати ще й метод `__radd__()`. Програмний код цього методу міг би бути таким:

```
def __radd__(self, x):
    return self+x
```

Розділ 7. Продовжуємо знайомство з ООП

У такому разі ми в тілі методу `__radd__()` неявно викликаємо метод `__add__()`, оскільки саме його буде залучено під час обчислення виразу `self+x`, у якому до екземпляра класу додається число.

У цьому випадку ми визначаємо операції додавання екземпляра до числа і числа до екземпляра так, що результати ідентичні. Але необхідності в цьому не було — ми могли б визначити ці операції по-різному.

Операторів, які можна перевантажувати, досить багато. Їх зазвичай розбивають на три групи: математичні оператори, двійкові (побітові) оператори й оператори порівняння. Далі в таблиці 7.3 перераховано перевантажувані математичні оператори. При цьому ми використовуємо такі позначення: через `obj` позначено екземпляр класу, в якому перевантажується метод, а інший операнд (якщо такий є) позначено через `x`.

Таблиця 7.3. Перевантажувані математичні оператори

Метод	Оператор	Приклад	Опис
<code>__add__()</code>	+	<code>obj+x</code>	Додавання до екземпляра класу <code>obj</code> операнда <code>x</code>
<code>__radd__()</code>	+	<code>x+obj</code>	Додавання до операнда <code>x</code> екземпляра класу <code>obj</code>
<code>__iadd__()</code>	<code>+=</code>	<code>obj+=x</code>	Додавання до екземпляра класу <code>obj</code> операнда <code>x</code> і присвоювання значення змінній екземпляра <code>obj</code>
<code>__sub__()</code>	-	<code>obj-x</code>	Віднімання від екземпляра класу <code>obj</code> операнда <code>x</code>
<code>__rsub__()</code>	-	<code>x-obj</code>	Віднімання від операнда <code>x</code> екземпляра класу <code>obj</code>
<code>__isub__()</code>	<code>-=</code>	<code>obj-=x</code>	Віднімання від екземпляра класу <code>obj</code> операнда <code>x</code> і присвоювання результату змінній екземпляра класу <code>obj</code>

Метод	Оператор	Приклад	Опис
<code>__mul__()</code>	*	<code>obj*x</code>	Обчислення добутку екземпляра класу <code>obj</code> і операнда <code>x</code>
<code>__rmul__()</code>	*	<code>x*obj</code>	Обчислення добутку операнда <code>x</code> і екземпляра класу <code>obj</code>
<code>__imul__()</code>	<code>*=</code>	<code>obj *= x</code>	Множення екземпляра класу <code>obj</code> на операнд <code>x</code> і присвоювання результутату змінній екземпляра класу <code>obj</code>
<code>__truediv__()</code>	/	<code>obj/x</code>	Ділення екземпляра класу <code>obj</code> на операнд <code>x</code>
<code>__rtruediv__()</code>	/	<code>x/obj</code>	Ділення операнда <code>x</code> на екземпляр класу <code>obj</code>
<code>__itruediv__()</code>	<code>/=</code>	<code>obj /= x</code>	Ділення екземпляра класу <code>obj</code> на операнд <code>x</code> і присвоювання результутату змінній екземпляра класу <code>obj</code>
<code>__floordiv__()</code>	//	<code>obj//x</code>	Цілочислове ділення екземпляра класу <code>obj</code> на операнд <code>x</code>
<code>__rfloordiv__()</code>	//	<code>x//obj</code>	Цілочислове ділення операнда <code>x</code> на екземпляр класу <code>obj</code>
<code>__ifloordiv__()</code>	<code>//=</code>	<code>obj //= x</code>	Цілочислове ділення екземпляра класу <code>obj</code> на операнд <code>x</code> і присвоювання результутату змінній екземпляра класу <code>obj</code>
<code>__mod__()</code>	<code>%</code>	<code>obj%x</code>	Залишок від ділення екземпляра класу <code>obj</code> на операнд <code>x</code>

Розділ 7. Продовжуємо знайомство з ООП

Метод	Оператор	Приклад	Опис
<code>__rmod__()</code>	<code>%</code>	<code>x%obj</code>	Залишок від ділення операんだ <code>x</code> на екземпляр класу <code>obj</code>
<code>__imod__()</code>	<code>%=</code>	<code>obj %= x</code>	Залишок від ділення екземпляра класу <code>obj</code> на операнд <code>x</code> і присвоювання результату змінній екземпляра класу <code>obj</code>
<code>__pow__()</code>	<code>**</code>	<code>obj ** x</code>	Піднесення екземпляра класу <code>obj</code> до степеня операнда <code>x</code>
<code>__rpow__()</code>	<code>**</code>	<code>x ** obj</code>	Піднесення операнда <code>x</code> до степеня екземпляра класу <code>obj</code>
<code>__ipow__()</code>	<code>**=</code>	<code>obj **= x</code>	Піднесення екземпляра класу <code>obj</code> до степеня операнда <code>x</code> і присвоювання результату змінній екземпляра класу <code>obj</code>
<code>__neg__()</code>	<code>-</code>	<code>-obj</code>	Застосування унарного (один operand) оператора «мінус» до екземпляра класу <code>obj</code>
<code>__pos__()</code>	<code>+</code>	<code>+obj</code>	Застосування унарного (один operand) оператора «плюс» до екземпляра класу <code>obj</code>
<code>__abs__()</code>	модуль	<code>abs(obj)</code>	Обчислення модуля екземпляра класу <code>obj</code>

У таблиці 7.4 перераховано перевантажувані побітові оператори. Як і в попередньому випадку, через `obj` позначено екземпляр класу, в якому перевантажується оператор, а через `x` — інший operand.

Таблиця 7.4. Перевантажувані побітові оператори

Метод	Оператор	Приклад	Опис
<code>__invert__()</code>	<code>~</code>	<code>~obj</code>	Побітова <i>інверсія</i> екземпляра класу <code>obj</code>
<code>__and__()</code>	<code>&</code>	<code>obj & x</code>	<i>Побітове i</i> для екземпляра класу <code>obj</code> і операнда <code>x</code>
<code>__rand__()</code>	<code>&</code>	<code>x & obj</code>	<i>Побітове i</i> для операнда <code>x</code> і екземпляра класу <code>obj</code>
<code>__iand__()</code>	<code>&=</code>	<code>obj &= x</code>	<i>Побітове i</i> для екземпляра класу <code>obj</code> і операнда <code>x</code> із присвоюванням результути змінній екземпляра класу <code>obj</code>
<code>__or__()</code>	<code> </code>	<code>obj x</code>	<i>Побітове або</i> для екземпляра класу <code>obj</code> і операнда <code>x</code>
<code>__ror__()</code>	<code> </code>	<code>x obj</code>	<i>Побітове або</i> для операнда <code>x</code> і екземпляра класу <code>obj</code>
<code>__ior__()</code>	<code> =</code>	<code>obj = x</code>	<i>Побітове або</i> для екземпляра класу <code>obj</code> і операнда <code>x</code> із присвоюванням результути змінній екземпляра класу <code>obj</code>
<code>__xor__()</code>	<code>^</code>	<code>obj ^ x</code>	<i>Побітове виключне або</i> для екземпляра класу <code>obj</code> і операнда <code>x</code>
<code>__rxor__()</code>	<code>^</code>	<code>x ^ obj</code>	<i>Побітове виключне або</i> для операнда <code>x</code> і екземпляра класу <code>obj</code>
<code>__ixor__()</code>	<code>^=</code>	<code>obj ^= x</code>	<i>Побітове виключне або</i> для екземпляра класу <code>obj</code> і операнда <code>x</code> із присвоюванням результути змінній екземпляра класу <code>obj</code>

Метод	Оператор	Приклад	Опис
<code>__lshift__()</code>	<code><<</code>	<code>obj<<x</code>	Зсув ліворуч для екземпляра класу <code>obj</code> на операнд <code>x</code>
<code>__rlshift__()</code>	<code><<</code>	<code>x<<obj</code>	Зсув ліворуч для операнда <code>x</code> на екземпляр класу <code>obj</code>
<code>__ilshift__()</code>	<code><<=</code>	<code>obj<<=x</code>	Зсув ліворуч для екземпляра класу <code>obj</code> на операнд <code>x</code> із присвоюванням результату змінній екземпляра класу <code>obj</code>
<code>__rshift__()</code>	<code>>></code>	<code>obj>>x</code>	Зсув праворуч для екземпляра класу <code>obj</code> на операнд <code>x</code>
<code>__rrshift__()</code>	<code>>></code>	<code>x>>obj</code>	Зсув праворуч для операнда <code>x</code> на екземпляр класу <code>obj</code>
<code>__irshift__()</code>	<code>>>=</code>	<code>obj>>=x</code>	Зсув праворуч для екземпляра класу <code>obj</code> на операнд <code>x</code> із присвоюванням результату змінній екземпляра класу <code>obj</code>

Таблиця 7.5 містить відомості про перевантажувані оператори порівняння (`obj` — екземпляр класу, а `x` — інший операнд).

Таблиця 7.5. Перевантажувані оператори порівняння

Метод	Оператор	Приклад	Опис
<code>__eq__()</code>	<code>==</code>	<code>obj==x</code>	Рівність екземпляра класу <code>obj</code> і операнда <code>x</code>
<code>__ne__()</code>	<code>!=</code>	<code>obj != x</code>	Нерівність екземпляра класу <code>obj</code> і операнда <code>x</code>
<code>__lt__()</code>	<code><</code>	<code>obj < x</code>	Екземпляр класу <code>obj</code> менший від операнда <code>x</code>
<code>__gt__()</code>	<code>></code>	<code>obj > x</code>	Екземпляр класу <code>obj</code> більший за операнд <code>x</code>

Метод	Оператор	Приклад	Опис
<code>__le__()</code>	<code><=</code>	<code>obj <= x</code>	Екземпляр класу <code>obj</code> не більший за операнд <code>x</code>
<code>__ge__()</code>	<code>>=</code>	<code>obj >= x</code>	Екземпляр класу <code>obj</code> не менший від операнда <code>x</code>
<code>__contains__()</code>	<code>in</code>	<code>x in obj</code>	Операнд <code>x</code> входить до екземпляра класу <code>obj</code>

Операторів, що підходять для перевантаження, досить багато. Тут важливо зрозуміти просту річ: ми, в принципі, можемо перевантажувати оператори як завгодно (у певних межах, звичайно). Важливо тільки те, який це оператор: бінарний (той, якому потрібно два операнди) або унарний (в оператора — один операнд). Приклад, у якому виконується перевантаження деяких із перерахованих вище операторів, наведено в лістингу 7.16.



Далі описується клас для реалізації такого математичного об'єкта, як **вектор**. Якщо абстрагуватися від геометричної інтерпретації вектора, то з деякою натяжкою можна стверджувати, що вектор — це набір із трьох параметрів, які називаються координатами вектора. Із векторами можна виконувати певні операції. Так, якщо є два вектори $\vec{a} = (a_1, a_2, a_3)$ і $\vec{b} = (b_1, b_2, b_3)$, то можна обчислити суму векторів $\vec{c} = \vec{a} + \vec{b}$. За означенням, це вектор $\vec{c} = (c_1, c_2, c_3)$ із координатами, що дорівнюють сумі координат векторів, що додаються: $c_k = a_k + b_k$ (індекс $k = 1, 2, 3$). Аналогічно обчислюється різниця векторів: $\vec{c} = \vec{a} - \vec{b}$, причому координати вектора $\vec{c} = (c_1, c_2, c_3)$ дорівнюють $c_k = a_k - b_k$ (індекс $k = 1, 2, 3$).

Скалярним добутком векторів $\vec{a} = (a_1, a_2, a_3)$ і $\vec{b} = (b_1, b_2, b_3)$ називається число $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$, тобто сума добутків відповідних координат векторів. При множенні вектора на число всі його координати помножуються на це число: якщо $\vec{a} = (a_1, a_2, a_3)$, то $\lambda \vec{a} = (\lambda a_1, \lambda a_2, \lambda a_3)$. Аналогічно вектор ділиться на число: треба поділити кожну координату вектора на це число — за визначенням $\frac{\vec{a}}{\lambda} = \left(\frac{a_1}{\lambda}, \frac{a_2}{\lambda}, \frac{a_3}{\lambda} \right)$. Існує така характеристика, як модуль вектора. Модуль вектора — це число, яке дорівнює квадратному кореню зі

скалярного добутку вектора на самого себе. Для вектора $\vec{a} = (a_1, a_2, a_3)$ модуль $|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}} = \sqrt{a_1^2 + a_2^2 + a_3^2}$.

Лістинг 7.16. Переображення операторів

```
# Функція для обчислення квадратного кореня
from math import sqrt
# Клас для реалізації вектора
class Vector:
    # Конструктор
    def __init__(self, x=0, y=0, z=0):
        # Полям присвоюються значення
        self.x=x
        self.y=y
        self.z=z
    # Метод для зведення до текстового типу
    def __str__(self):
        # Текстове значення
        txt=<">+str(self.x)+<" | "
        txt+=str(self.y)+<" | "
        txt+=str(self.z)+>">
        # Результат методу
        return txt
    # Додавання векторів
    def __add__(self,obj):
        # Створюється екземпляр класу
        t=Vector()
        # Значення полів екземпляра
        t.x=self.x+obj.x
        t.y=self.y+obj.y
        t.z=self.z+obj.z
        # Результат методу
        return t
    # Додавання векторів із присвоюванням
    def __iadd__(self,obj):
        # Змінюємо екземпляр
        self=self+obj
```

```
# Результат методу
    return self

# Множення вектора на вектор
# або вектора на число
def __mul__(self,p):
    # Перевіряємо тип аргументу
    if type(p)==Vector:
        # Добуток векторів
        res=self.x*p.x+self.y*p.y+self.z*p.z
        # Результат методу
        return res
    # Добуток вектора на число
    else:
        self.x*=p
        self.y*=p
        self.z*=p
        # Результат методу
        return self

# Множення числа на вектор
def __rmul__(self,p):
    # Результат методу
    return self*p

# Мінус перед вектором
def __neg__(self):
    # Результат методу
    return Vector(-self.x,-self.y,-self.z)

# Різниця векторів
def __sub__(self,obj):
    # Результат методу
    return -obj+self

# Різниця векторів із присвоюванням
def __isub__(self,obj):
    # Змінюємо об'єкт
    self=-obj+self
    # Результат методу
    return self
```

```

# Модуль вектора
def __abs__(self):
    # Результат методу
    return sqrt(self*self)

# Ділення вектора на число
def __truediv__(self,p):
    # Результат методу
    return self*(1/p)

# Рівність векторів
def __eq__(self,obj):
    # Якщо рівні значення полів
    if self.x==obj.x and self.y==obj.y and self.z==obj.z:
        return True
    # Якщо значення полів різні
    else:
        return False

# Нерівність векторів
def __ne__(self,obj):
    # Результат методу
    return not self==obj

# Один вектор "менший" від іншого
def __lt__(self,obj):
    # Якщо модуль першого вектора менший
    # від модуля другого вектора
    if abs(self)<abs(obj):
        return True
    # Якщо це не так
    else:
        return False

# Один вектор "більший" за інший
def __gt__(self,obj):
    # Якщо модуль першого вектора більший
    # за модуль другого вектора
    if abs(self)>abs(obj):
        return True

```

Phyton

```
# Якщо це не так
else:
    return False

# Один вектор "не більший" за інший
def __le__(self,obj):
    # Якщо модуль першого вектора не більший
    # за модуль другого вектора
    if abs(self)<=abs(obj):
        return True
    # Якщо це не так
    else:
        return False

# Один вектор "не менший" від іншого
def __ge__(self,obj):
    # Якщо модуль першого вектора не менший
    # від модуля другого вектора
    if abs(self)>=abs(obj):
        return True
    # Якщо це не так
    else:
        return False

# Побітова інверсія для вектора
def __invert__(self):
    # Присвоюються значення полям
    self.x=10-self.x
    self.y=10-self.y
    self.z=10-self.z
    # Результат методу
    return self

# Зсув уліво (екземпляр класу - перший операнд)
def __lshift__(self,n):
    # Виконуються циклічні перестановки полів
    for i in range(n):
        self.x,self.y,self.z=self.y,self.z,self.x
    # Результат методу
    return self
```

```

# Зсув уліво (екземпляр класу - другий операнд)
def __rlshift__(self,n):
    # Результат методу
    return self>>n

# Зсув управо (екземпляр класу - перший операнд)
def __rshift__(self,n):
    # Виконується циклічна перестановка полів
    for i in range(n):
        self.x,self.y,self.z=self.z,self.x,self.y
    # Результат методу
    return self

# Зсув управо (екземпляр класу - другий операнд)
def __rrshift__(self,n):
    # Результат методу
    return self<<n

# Вектори
print("Вектори:")
a=Vector(1,2,-1)
b=Vector(1,-1,3)
c=~Vector(9,8,11)
print("a =",a)
print("b =",b)
print("c =",c)
# Обчислення модуля
print("Модуль вектора.")
print("|a| =",abs(a))
print("|b| =",abs(b))
print("|c| =",abs(c))
# Порівняння векторів
print("Порівняння векторів.")
print("a==b ->",a==b)
print("a!=b ->",a!=b)
print("a==c ->",a==c)
print("a<b ->",a<b)
print("a>b ->",a>b)
print("a<=c ->",a<=c)
print("a>=c ->",a>=c)

```

Phyton

```
# Операції з векторами
print("Сума векторів:")
print("a+b =", a+b)
c+=a
print("c+=a ->", c)
print("Різниця векторів:")
print("a-b =", a-b)
c-=a
print("c-=a ->", c)
print("Добуток векторів:")
print("a*b =", a*b)
print("Множення й ділення вектора на число:")
print("a*3 =", a*3)
print("a =", a)
print("2*b =", 2*b)
print("b =", b)
print("-b =", -b)
print("b =", b)
print("a/3 =", a/3)
print("a =", a)
print("Циклічні перестановки:")
v=Vector(1,2,3)
print("v =", v)
print("v<<1 =", v<<1)
print("v>>1 =", v>>1)
print("2>>v =", 2>>v)
print("2<<v =", 2<<v)
```

Результат виконання програмного коду такий:



Результат виконання програми (з лістингу 7.16)

Вектори:

```
a = <1|2|-1>
b = <1|-1|3>
c = <1|2|-1>
```

Розділ 7. Продовжуємо знайомство з ООП

Модуль вектора.

$|a| = 2.449489742783178$

$|b| = 3.3166247903554$

$|c| = 2.449489742783178$

Порівняння векторів.

$a==b \rightarrow False$

$a!=b \rightarrow True$

$a==c \rightarrow True$

$a < b \rightarrow True$

$a > b \rightarrow False$

$a <= c \rightarrow True$

$a >= c \rightarrow True$

Сума векторів:

$a+b = <2|1|2>$

$c+a \rightarrow <2|4|-2>$

Різниця векторів:

$a-b = <0|3|-4>$

$c-a \rightarrow <1|2|-1>$

Добуток векторів:

$a*b = -4$

Множення й ділення вектора на число:

$a*3 = <3|6|-3>$

$a = <3|6|-3>$

$2*b = <2|-2|6>$

$b = <2|-2|6>$

$-b = <-2|2|-6>$

$b = <2|-2|6>$

$a/3 = <1.0|2.0|-1.0>$

$a = <1.0|2.0|-1.0>$

Циклічні перестановки:

$v = <1|2|3>$

$v<<1 = <2|3|1>$

$v>>1 = <1|2|3>$

$2>>v = <3|1|2>$

$2<<v = <1|2|3>$

Оскільки при обчисленні модуля вектора ми збираємося використовувати функцію `sqrt()` для знаходження квадратного кореня, починаємо програмний код з інструкції `from math import sqrt` імпорту цієї функції з модуля `math`.

Основу програмного коду становить опис класу `Vector`, через який реалізуються вектори. У класу є конструктор. Якщо під час створення екземпляра класу конструктору передаються три аргументи, то це будуть координати вектора, який реалізується у вигляді екземпляра класу. Якщо під час створення екземпляра аргументи не передаються, то це відповідає нульовим координатам вектора.

У класі описано спеціальний метод `__str__()`, тож екземпляри класу можна передавати як аргумент функції `print()`.

Для реалізації операції додавання векторів (тобто додавання екземплярів класу) описуються методи `__add__()` (додавання екземплярів) і `__iadd__()` (додавання екземплярів із присвоюванням).



Метод `__add__()` викликається у випадку, коли першим операндом є екземпляр класу. У принципі, для випадку, коли екземпляр класу є другим операндом, треба було б описати метод `__radd__()`. Але оскільки в такому разі обидва операнди будуть екземплярами класу (ми розглядаємо тільки такі випадки), то немає необхідності описувати метод `__radd__()`.

У тілі методу `__add__()` створюється екземпляр `t` класу `Vector`, а потім значення полів цього екземпляра обчислюються як сума полів екземплярів-операндів. Після цього екземпляр `t` повертається як результат методу `__add__()`.

У тілі методу `__iadd__()` виконується команда `self=self+obj`, яка до екземпляра `self` (перший операнд) додає екземпляр `obj` (другий операнд), і результат записується в змінну `self`. Ця змінна повертається як результат методу.



Під час обчислення виразу `self=self+obj` у правій частині обчислюється сума двох екземплярів `self` і `obj`. Для обчислення суми екземплярів викликається метод `__add__()`. Таким чином, описуючи один спеціальний метод, ми використовуємо неявний виклик іншого описаного в класі спеціального методу.

Метод `__mul__()` призначений для виконання множення вектора на вектор і вектора на число. У першому випадку йдеться про множення двох екземплярів, а в другому — про обчислення добутку екземпляра на число. Залежно від того, до якого типу належить другий операнд, результат буде різним. Другий операнд позначено в списку аргументів методу `__mul__()` як `p`. У тілі методу в умовному операторі перевіряється умова `type(p) == Vector`, яка істинна, якщо змінна `p` посилається на екземпляр класу `Vector`. У такому разі обчислюється значення виразу `self.x*p.x+self.y*p.y+self.z*p.z` (сума добутків відповідних полів екземплярів `self` і `p`), обчислене значення записується в змінну `res`, і ця змінна повертається як результат методу.

Якщо умова `type(p) == Vector` хибна, ми припускаємо, що аргумент `p` — це число. На це число помножуються всі поля екземпляра `self`, після чого посилання на екземпляр `self` повертається результатом методу.



Таким чином, при множенні екземпляра на екземпляр, вихідні екземпляри не змінюються, а результат добутку — число. Якщо ж екземпляр помножується на число, то змінюється вихідний екземпляр.

Також у класі `Vector` описано метод `__rmul__()`. Цей метод викликається, якщо другий операнд при множенні є екземпляром класу `Vector`, а перший операнд таким екземпляром не є (ми виходимо з того, що це число). Отже, метод «виходить на сцену» при спробі множення числа (аргумент `p` методу) на екземпляр класу `Vector` (аргумент `self` методу). У тілі методу виконується лише одна інструкція `return self*p`, яка результатом повертає добуток екземпляра `self` на числове значення `p`. Фактично, ми операцію множення числа на екземпляр класу визначаємо як множення екземпляра на число.

Метод `__neg__()` викликається, коли перед екземпляром указується знак *мінус*. Причому в цьому випадку йдеться про мінус як про унарну операцію. Цей метод ми описуємо так, що екземпляр класу змінюватися не буде, а як результат повертається знову створений екземпляр, поля якого отримуємо з полів вихідного екземпляра шляхом множення на -1 .

Різниця векторів обчислюється методом `__sub__()`. Результатом методу є значення виразу $-obj + self$, у якому унарний мінус застосовується до екземпляра `obj` і до отриманого в результаті цієї операції екземпляра додається екземпляр `self`. Дещо схоже відбувається у тілі методу `__isub__()`, але тільки тепер результат присвоюється змінній `self`.

Модуль вектора обчислюється методом `__abs__()` як значення виразу `sqrt(self * self)` (квадратний корінь із добутку екземпляра `self` на себе самого).

Ділення вектора на число реалізується через метод `__truediv__()`. Тут ми входимо з того, що ділення екземпляра `self` на число `p` — це те саме, що множення екземпляра `self` на число `1/p`. Тому результат методу обчислюється виразом `self * (1/p)`.

Під час порівняння екземплярів на предмет рівності викликається метод `__eq__()`. Метод описано так, що значення `True` повертається, якщо збігаються значення полів порівнюваних екземплярів. Порівняння екземплярів на предмет нерівності реалізується завдяки методу `__ne__()`. Результат методу (вираз `not self == obj`, де `self` і `obj` — аргументи методу) отримується логічним запереченням результату порівняння екземплярів на предмет рівності.

При порівнянні екземплярів за допомогою операторів *більше*, *менше*, *не більше і не менше* (відповідно методи `__gt__()`, `__lt__()`, `__le__()` і `__ge__()`) насправді порівнюються модулі екземплярів.

У класі `Vector` описуються й деякі методи для побітових операторів. Так, *побітова інверсія* для екземпляра класу виконується за допомогою методу `__invert__()`: «інвертований» екземпляр змінюється так, що нові значення полів отримують шляхом віднімання з числа `10` поточних значень полів екземпляра.

Крім цього, описуються методи `__lshift__()`, `__rlshift__()`, `__rshift__()` і `__rrshift__()`. Ми припускаємо, що один із аргументів у кожному з цих методів — екземпляр класу `Vector`, а інший операнд — це ціле число. Під час виклику методів (а це відбувається за використання операторів `<< i >>`) виконується циклічна перестановка значень полів екземпляра класу. Кількість послідовних перестановок визначається значенням числового операнда. Напрямок перестановки (ліворуч або праворуч) визначається оператором і порядком аргументів. Перестановки вліво відбуваються для команд вигляду `екземпляр<<число і число>>екземпляр`, а перестановка вправо виконується для команд вигляду `екземпляр>>число і число<<екземпляр`. Наприклад, у тілі методу `__lshift__()` запускається оператор циклу, в якому число ітерацій визначається числовим аргументом, а за кожний цикл виконується команда множинного присвоювання `self.x, self.y, self.z=self.y, self.z, self.x`. У результаті поле `x` отримує значення поля `y`, поле `y` — значення поля `z`, а поле `z` — значення поля `x`.



При обчисленні виразу `self.x, self.y, self.z=self.y, self.z, self.x`, варто врахувати, що спочатку обчислюються значення у правій частині, а потім кожне з них присвоюється відповідній змінній у лівій частині.

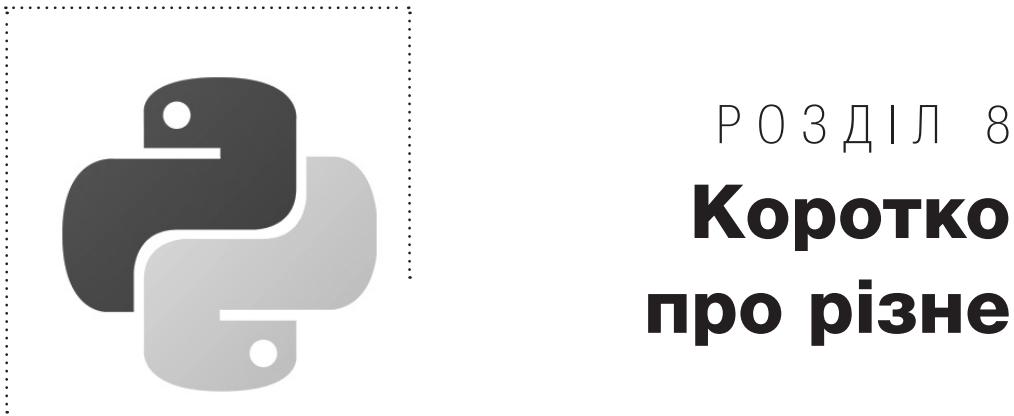
Поза тілом класу група команд дозволяє перевірити функціональність описаних у класі `Vector` спеціальних методів.

Резюме

Покличте репортерів: у нас демократія померла!

З телесеріалу «Альф»

- Наслідування дозволяє створювати нові класи на основі вже існуючих класів. Новий створений клас називається похідним, а той клас, на основі якого створюється похідний клас, називається базовим. Поля та методи базового класу наслідуються в похідному класі.
- При створенні похідного класу базовий клас указується в круглих дужках після імені похідного класу. Базових класів може бути декілька. У цьому випадку їхні імена в круглих дужках розділяються комами. Похідний клас може, у свою чергу, бути базовим для іншого класу.
- Методи, наслідувані в похідному класі з базового, можна перевизначати (описувати заново в похідному класі).
- Існує група спеціальних полів і методів, які дозволяють отримувати важливу інформацію про клас. Також за допомогою спеціальних методів можна перевантажувати оператори (такі, як оператор додавання, віднімання, множення, побітової інверсії або оператор порівняння), щоб використовувати їх у роботі з екземплярами класу.



РОЗДІЛ 8

Коротко про різне

І на ці гроші ми купимо багато
благ... і гордо от них откажемся.
Может быть.

«Шоу Довгоносиків»

Ми вже розглянули багато питань щодо основних синтаксичних конструкцій мови Python, дізналися про відмінність між списками, кортежами, множинами й словниками, познайомилися з інструкціями керування (такими, як умовний оператор або оператор циклу), з'ясували, як описуються функції, класи, а також як створюються екземпляри класів. Обговорювалися й інші теми та питання. Втім, було б наївно думати, що отримані відомості є вичерпними. Зрозуміло, що багато залишилося «за бортом». Зараз настав час зазирнути за цей самий «борт». Якщо конкретніше, то ми в цьому розділі розглянемо деякі питання, що стосуються, в основному, роботи з функціями і класами (екземплярами класу). Також розглянемо невеликі приклади — показові з точки зору методики програмування мовою Python.

Функції зі змінною кількістю аргументів

Кульмінація апогею нашого апофейозу.

«Шоу Довгоносиків»

У Python існує можливість описувати (створювати) функції, яким може передаватися, від виклику до виклику, різна (довільна) кількість аргументів. За великим рахунком, ідеться про те, що під час виклику функції ми можемо зазначати ту чи іншу кількість аргументів, залежно від потреб. Зі схожою ситуацією ми вже зустрічалися, коли мали справу з функціями, аргументи яких мають значення за замовчуванням. Наприклад, якщо у функції описано п'ять аргументів, і в кожного є значення за замовчуванням, то під час виклику функції її можна не передавати аргументи зовсім або вказати від одного до п'яти аргументів. Але в такому підході є суттєвий недолік: при описі всі аргументи необхідно перераховувати явно. Це, як мінімум, накладає обмеження на максимальну кількість елементів, які можна передати функції. Крім того, часто важливі значення мають не тільки самі аргументи, які передані функції, а й їхня кількість. Тобто кількість аргументів функції може відігравати роль неявного параметра або аргументу, що використовується при виконанні функції або обчисленні її результату. У такому разі, підхід, заснований на використанні значень за замовчуванням для аргументів функції, далеко не найоптимальніший.

Є інший варіант — передавати аргументи функції у вигляді списку. Але це також не завжди зручно, оскільки в цьому випадку аргумент насправді один і оброблювати його треба як список. Організувати таку обробку досить просто, але основні труднощі виникають при передачі аргументів

під час виклику функції, оскільки ці аргументи попередньо доведеться організувати у вигляді списку. Хоча, звичайно, і ця задача не є важкою. Приклад такого підходу проілюстровано в лістингу 8.1.

Лістинг 8.1. Аргумент функції - список

```
# Функція з аргументом - списком
def show_me(args=[]):
    # Початкове значення індексу
    i=0
    print("Аргумент (и) функції:")
    for s in args:
        # Нове значення індексу
        i+=1
        # Відображується елемент списку
        print(str(i)+"-й 'аргумент':",s)
    print("Виконання функції закінчено.")

# Приклади виклику функції
show_me()
show_me([100])
# Список - аргумент функції
nums=[10,55,2+3j,0.123]
show_me(nums)
show_me(["Python","Java"])
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 8.1)

```
Аргумент (и) функції:
Виконання функції закінчено.

Аргумент (и) функції:
1-й 'аргумент': 100
Виконання функції закінчено.

Аргумент (и) функції:
1-й 'аргумент': 10
2-й 'аргумент': 55
3-й 'аргумент': (2+3j)
```

```

4-й 'аргумент': 0.123
Виконання функції закінчено.

Аргумент (и) функції:
1-й 'аргумент': Python
2-й 'аргумент': Java
Виконання функції закінчено.

```

У цьому випадку ми описуємо функцію `show_me()`, в якої один аргумент, причому в цього аргументу є значення за замовчуванням — порожній список. Сам аргумент, як ми припускаємо, також є списком. У будь-якому разі, оброблюємо ми його саме як список. Зокрема, у тілі функції запускається оператор циклу, і в цьому операторі циклу змінна `s` «пробігає» значення зі списку `args` (аргумент функції). Індексна змінна і за кожний цикл збільшує своє значення на одиницю, нумеруючи тим самим елементи списку `args`. Операції з цими елементами виконуються прості: вони відображуються у вікні виводу. При цьому кожний елемент ми называемо «аргументом» — як нагадування, що список `args`, переданий у функцію `show_me()`, «імітує» набір аргументів, і кількість цих аргументів не фіксована.

Потім ідуть приклади виклику функції `show_me()`: без аргументів (це можливо, оскільки в описі функції для її аргументу-списку задано значення за замовчуванням), із аргументом-списком із одного елемента, із чотирьох елементів і з двох елементів. У кожному з цих випадків параметри, які передаються у функцію, повинні бути організовані у вигляді списку (спісок створюється попередньо, і посилання на нього присвоюється змінній або список безпосередньо передається аргументом функції).



Формально, у функції `show_me()` один аргумент (і ми виходимо з того, що цей аргумент — список). Тому під час виклику функції їй треба передати один аргумент. Щоб можна було викликати функцію без передачі аргументу, ми задаємо значення аргументу за замовчуванням (порожній список).

Під час виклику функції `show_me()`, якщо їй не передається аргумент, у тілі функції в операторі циклу перебираються елементи порожнього списку. Але оскільки в порожньому списку елементів немає, то й перебирати особливо немає чого. Як наслідок, жодна з команд у тілі оператора циклу не виконується.

Phyton

У принципі, немає нічого поганого в тому, щоб використовувати означений вище підхід (із передачею списку аргументом функції), але є й інші можливості. Як ілюстрацію розглянемо програмний код у лістингу 8.2. Порівняно з попереднім прикладом тут унесено мінімальні зміни (які, проте, мають суттєві наслідки).

Лістинг 8.2. Функція зі змінною кількістю аргументів

```
# Функція з довільною кількістю аргументів.  
# Єдиний формальний аргумент оброблюється  
# як список  
def show_me(*args):  
    # Початкове значення індексу  
    i=0  
    print("Аргумент (и) функції:")  
    for s in args:  
        # Нове значення індексу  
        i+=1  
        # Відображується елемент списку  
        print(str(i)+"-й аргумент:",s)  
    print("Виконання функції закінчено.")  
# Приклади виклику функції  
show_me()  
show_me(100)  
# Аргумент функції формується на основі списку  
nums=[10,55,2+3j,0.123]  
show_me(*nums)  
show_me("Python","Java")
```

Результат виконання програмного коду виглядатиме так:

Результат виконання програми (з лістингу 8.2)

```
Аргумент (и) функції:  
Виконання функції закінчено.  
Аргумент (и) функції:  
1-й аргумент: 100  
Виконання функції закінчено.
```

Аргумент (и) функції:

1-й аргумент: 10

2-й аргумент: 55

3-й аргумент: (2+3j)

4-й аргумент: 0.123

Виконання функції закінчено.

Аргумент (и) функції:

1-й аргумент: Python

2-й аргумент: Java

Виконання функції закінчено.

Усе, що ми зробили, — це в описі функції `show_me()` перед аргументом `args` поставили зірочку `*`. При цьому в тілі функції змінна `args` оброблюється як список (а якщо точніше, як *кортеж* — але в цьому разі відмінність невідчутна). А ось під час виклику функції аргументи їй можуть передаватися в будь-якій кількості (включаючи «екзотичний» випадок, коли аргументів немає зовсім). У вигляді списку аргументи оформлювати не треба. Більше того, якщо в нас є список (як, наприклад, `nums=[10, 55, 2+3j, 0.123]`), то його можна передати аргументом функції `show_me()`, вказавши перед іменем списку зірочку `*` (як, наприклад, у команді `show_me(*nums)`). Ефект буде такий, як начеб аргументами функції передавалися елементи списку.

Отже, щоб описати функцію з довільною кількістю аргументів, формально у функції зазначається один аргумент, перед яким повинна бути зірочка `*`. При цьому в тілі функції аргумент оброблюється як список. Під час виклику функції аргументи передаються як звичайно, без формування списку. Якщо під час виклику функції їй передати аргументом список і перед цим списком поставити зірочку `*`, то функцію буде викликано з аргументами, які є елементами цього списку.



Останнє зауваження стосується не лише функцій із довільною кількістю аргументів, а й усіх функцій. Наприклад, еквівалентом команди `pow(*[10, 2])` є команда `pow(10, 2)`. Нагадаємо, що за допомогою функції `pow()` виконується піднесення до степеня, тому результат обох команд — це 10^2 , тобто число 100.

Ситуація може бути не такою очевидною, як у розглянутому прикладі. Варіантів досить багато, і ми зупинимося на найбільш значущих. Скажімо, нерідко трапляється, що необхідно описати функцію, в якій певна кількість аргументів повинна бути передана під час виклику в обов'язковому порядку. Якщо так, то функція повинна містити явний опис необхідних аргументів (кожний із них у послідовності аргументів позначається окремою змінною), а всі інші аргументи (кількість яких не фіксована) позначаються одним аргументом із зірочкою. Приклад такої ситуації подано в лістингу 8.3.

Лістинг 8.3. Обов'язкові і необов'язкові аргументи

```
# Функція з двома, як мінімум, аргументами
def show_me(first,second,*other):
    # Перший аргумент
    print("Перший аргумент:",first)
    # Другий аргумент
    print("Другий аргумент:",second)
    # Кількість необов'язкових аргументів
    n=len(other)
    print("Ще залишилося",n,"аргументи:")
    # Кортеж із необов'язкових аргументів
    print(other)
# Виклик функції
show_me(10,20,30,40,50)
```

У результаті виконання цього програмного коду у вікні виводу з'являються такі повідомлення:

Результат виконання програми (з лістингу 8.3)

```
Перший аргумент: 10
Другий аргумент: 20
Ще залишилося 3 аргументи:
(30, 40, 50)
```

У цьому випадку у функції `show_me()` перші два аргументи `first` і `second` описано без зірочки, тому це — звичайні аргументи. Вони обов'язково повинні бути вказані під час виклику функції. Третій аргумент функції,

який називається `other`, зазначено із зірочкою. Тому під час виклику функції `show_me()` перший переданий їй аргумент — це аргумент `first`, другий переданий функції аргумент — це аргумент `second`, а всі інші аргументи функції — це кортеж `other`. У тілі функції аргумент `other` оброблюється як кортеж (або як список — тут це не принципово). Так, командою `n=len(other)` обчислюється і в змінну `n` записується кількість елементів у кортежі `other`. Це, фактично, кількість тих необов'язкових аргументів, які передано функції `show_me()` під час виклику. Щоб продемонструвати, що `other` — це саме кортеж, у тілі функції виконується команда `print(other)`.

Викликається функція `show_me()` із п'ятьма аргументами: перші два — аргументи `first` і `second`, а три останні формують кортеж `other`.

Але це ще не все. Читач, мабуть, пам'ятає, що аргументи функції можна передавати не тільки простим перерахуванням через кому (як ми зазвичай і робимо). Ще для аргументів можна явно вказувати *ключ* або *ім'я* аргументу у форматі `аргумент=значення`. Перший спосіб передачі аргументів функції називається *позиційним*, а другий — *за ключем* або *за іменем*. Під час виклику спочатку зазначаються позиційні аргументи (передаються у строгій відповідності до порядку, в якому аргументи описано у функції), а вже потім ті, що передаються за ключем (можуть бути в довільному порядку, оскільки назву кожного аргументу зазначено явно). У Python існує можливість створювати функції з довільною кількістю аргументів, причому ці аргументи можна передавати не тільки позиційним способом, але й з явним зазначенням ключа. Для позначення останніх використовують формальний аргумент, перед яким розміщують подвійну зірочку `**`. Якщо точніше, то такий аргумент є словником, ключі якого — це назви аргументів, а значення словника — це значення аргументів функції. Скажімо, у наступному прикладі описано функцію `show_me()` із аргументами `first`, `second`, `*other` і `**byname`. Перші два аргументи `first` і `second` функції повинні бути передані обов'язково (позиційним способом або через ключ), аргумент `other` є кортежем зі списком тих аргументів (за винятком `first` і `second`), які передано функції позиційним способом, а аргумент `byname` є словником із ключами і значеннями аргументів (крім `first` і `second`), які передано функції через ключ. Опис функції `show_me()` і приклади її виклику наведено в лістингу 8.4.

Лістинг 8.4. Передача аргументів за ключем

```
# Функція з довільною кількістю аргументів
def show_me(first,second,*other,**byname):
    # Відображуються значення аргументів
    print("first ->",first)
    print("second ->",second)
    print("other ->",other)
    print("byname ->",byname)

# Приклади виклику функції
print("1-й спосіб виклику функції.")
show_me(10,20,30,40,50,60,70)
print("2-й спосіб виклику функції.")
show_me(10,20,50,60,70,third=30,fourth=40)
print("3-й спосіб виклику функції.")
show_me(10,20,third=30,fourth=40)
print("4-й спосіб виклику функції.")
show_me(second=20,first=10)
print("5-й спосіб виклику функції.")
show_me(first=10,second=20,third=30,fourth=40)
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 8.4)

```
1-й спосіб виклику функції.
first -> 10
second -> 20
other -> (30, 40, 50, 60, 70)
byname -> {}

2-й спосіб виклику функції.
first -> 10
second -> 20
other -> (50, 60, 70)
byname -> {'third': 30, 'fourth': 40}

3-й спосіб виклику функції.
first -> 10
second -> 20
```

```

other -> ()
byname -> {'third': 30, 'fourth': 40}
4-й спосіб виклику функції.
first -> 10
second -> 20
other -> ()
byname -> {}
5-й спосіб виклику функції.
first -> 10
second -> 20
other -> ()
byname -> {'third': 30, 'fourth': 40}
.....
```

Якщо функція викликається з передачею всіх аргументів позиційним способом (як у команді `show_me(10,20,30,40,50,60,70)`), то перше і друге значення — це значення аргументів `first` і `second`, а всі інші аргументи формують кортеж, що є значенням аргументу `other`. Аргумент `byname` у цьому випадку є порожнім словником.

У команді `show_me(10,20,50,60,70,third=30,fourth=40)` крім позиційних аргументів, є аргументи, передані за ключем (причому йдеться про аргументи `first` і `second`). Перші два значення — аргументи `first` і `second`, усі інші позиційні аргументи — елементи кортежу `other`, а словник `byname` формується на основі аргументів, переданих за ключем. Конкретніше, значення аргументів `first` і `second` — це 10 і 20, кортеж `other` складається з трьох елементів (50, 60, 70), а словник `byname` містить елементи зі значеннями 30 і 40 із ключами `third` і `fourth` відповідно. Схожа ситуація з командою `show_me(10,20,third=30,fourth=40)`, тільки кортеж `other` цього разу порожній.



Зверніть увагу, що аргументи `third` і `fourth` в описі функції `show_me()` відсутні. Іншими словами, під час виклику функції ми зазначаємо ключі, яких наче й нема. Однак таке можна робити — мова Python надає можливості з обробки подібних ситуацій.

У команді `show_me(second=20,first=10)` перші обов'язкові аргументи зазначено за ключем, а інших аргументів немає. Тому й кортеж `other`, і словник `byname` цього разу порожні.

Нарешті, у команді `show_me(first=10,second=20,third=30,fourth=40)` усі аргументи зазначено за ключем. Як наслідок, аргумент `other` є порожнім кортежем.



Якщо під час виклику функції аргументом їй передати словник, перед яким розмістити дві зірочки `**`, то цей словник буде автоматично перетворено на список іменованих аргументів: ключі словника служать назвами аргументів, а значення словника — значеннями аргументів функції.

Декоратори функцій і класів

Скоро прийде атомний дирокол
і всіх нас врятує.

«Шоу Довгоносиків»

Щоб зрозуміти суть і призначення *декораторів функцій*, зробимо деякі попередні зауваження. Так, результатом функції може бути функція.



Приклад функції, яка як результат повертає функцію, наведено в розділі 3 у пункті, присвяченому опису **лямбда-функцій**, а також у пункті, спеціально присвяченому цій темі.

Аргументом функції також може бути функція. Тому ніхто і ніщо не забороняє описати нам функцію, яка як аргумент приймає функцію і як результат повертає функцію. Для більшої конкретики позначимо через $f()$ деяку функцію, а $F()$ — це буде функція, яка як аргумент отримує функцію і як результат повертає функцію. Функції $F()$ як аргумент може бути передано функцією $f()$ (аргументом указується ім'я функції). Таким чином, вираз $F(f)$ є функцією. Результат цього виразу можна присвоїти як значення змінній, і ця змінна посилатися на об'єкт функції. Іншими словами, таку змінну можна буде розглядати як функцію. Мало того, цією змінною може бути f — ми можемо скористатися командою $f=F(f)$. У цій команді немає помилки. Її виконання приведе до перевизначення функції $f()$.



Команда `f=F(f)` виконується так. Спочатку змінна `f` посилається на діякий об'єкт, який визначає функцію `f()`. Під час виконання команди `f=F(f)` спочатку розраховується результат виразу `F(f)`. При цьому використовується поточне посилання на об'єкт функції `f()`. Результатом виразу `F(f)` є новий об'єкт, що визначає діяку функцію. Посилання на цю функцію записується в змінну `f`. Зовнішній ефект від цих дій зводиться до того, що змінюється функція `f()`.

Таким чином, за допомогою функції `F()` ми можемо виконати перетворення функції `f()`. Для цього достатньо скористатися командою `f=F(f)`. Такого ж ефекту можна досягти, якщо перед описом функції `f()` поставити інструкцію `@F` (знак @ є ім'я функції `F()`, на основі якої виконується перетворення). Тобто йдеться про інструкцію вигляду

```
@F
def f(аргументи):
    # тіло функції
```

Про інструкцію `@F` говорять, що це — *декоратор функції*. Приклад використання декоратора функцій наведено в лістингу 8.5.



Далі ми розглядаємо «математичний» приклад. Зокрема, нас цікавить вираз вигляду $\exp(-f(x)^2)$, де $f(x)$ — діяка функція. Якщо б нам треба було часто використовувати такого роду вирази (для різних функцій $f(x)$), то розумно було б залучити декоратор функцій. А саме, ми описуємо функцію $F(f)$ таку, що $F(f)(x) = \exp(-f(x)^2)$. Функцію $F(f)$ достатньо описати один раз. Потім, використовуючи її як декоратор, можемо визначати функції для обчислення виразів вигляду $\exp(-f(x)^2)$, при цьому визначаючи фактично тільки код для обчислення виразу $f(x)$.

Лістинг 8.5. Декоратор функцій

```
# Імпорт математичних функцій
from math import exp,sin,cos,tan
# Функція для використання в декораторі
def F(f):
    # Лямбда-функція (анонімна функція)
```

```

res=lambda x: exp(-f(x)**2)
    return res
# Функція для використання в декораторі
def Q(f):
    # Внутрішня функція
    def q(x):
        return tan(f(x))
    return q
# Функція з декоратором
@F # Декоратор
def f(x):
    return sin(x)
# Функція з декоратором
@F # Декоратор
def g(x):
    return cos(x)
# Функція з двома декораторами
@Q # Другий декоратор
@F # Перший декоратор
def h(x):
    return x
# Змінна
n=5
# Значення функцій у різних точках
print("Функція f():")
for i in range(n+1):
    z=i/n
    print(f(z),"->",exp(-sin(z)**2))
print("Функція g():")
for i in range(n+1):
    z=i/n
    print(g(z),"->",exp(-cos(z)**2))
print("Функція h():")
for i in range(n+1):
    z=i/n
    print(h(z),"->,tan(exp(-z**2)))

```

Нижче наведено результат виконання програмного коду:

■ Результат виконання програми (з лістингу 8.5)

Функція $f()$:

```
1.0 -> 1.0
0.961299270288799 -> 0.961299270288799
0.8592918618974276 -> 0.8592918618974276
0.7270055824268487 -> 0.7270055824268487
0.5977397854322518 -> 0.5977397854322518
0.49259230319603176 -> 0.49259230319603176
```

Функція $g()$:

```
0.36787944117144233 -> 0.36787944117144233
0.38268981631591364 -> 0.38268981631591364
0.4281193125221935 -> 0.4281193125221935
0.5060201050223136 -> 0.5060201050223136
0.6154508201347391 -> 0.6154508201347391
0.7468233644427067 -> 0.7468233644427067
```

Функція $h()$:

```
1.5574077246549023 -> 1.5574077246549023
1.4307602577401106 -> 1.4307602577401106
1.143266474503948 -> 1.143266474503948
0.8383239288289558 -> 0.8383239288289558
0.582285681136045 -> 0.582285681136045
0.38542559176909813 -> 0.38542559176909813
```

На початку програмного коду ми імпортуємо з модуля `math` функції: `exp()` — для обчислення експоненти, `sin()` — для обчислення синуса, `cos()` — для обчислення косинуса і `tan()` — для обчислення тангенса. Далі описується функція $F()$, у якій, як припускається, аргумент f — ім'я іншої функції. У тілі функції змінній `res` значенням присвоюється посилання на анонімну функцію (лямбда-функцію), об'єкт якої створюється інструкцією `lambda x: exp(-f(x)**2)`. Змінна `res` (посилання на об'єкт функції) повертається як результат функції $F()$.

Функція $Q()$ також повертає як результат функцію (тому може й буде використана в декораторі). У тілі функції описується внутрішня функція $q()$, яка результатом повертає вираз $\tan(f(x))$, де x — аргумент

функції `q()`, а `f` — аргумент функції `Q()`. Ім'я функції `q` (посилання на об'єкт функції `q()`) повертається результатом функції `Q()`.



Отже, результатом виразу `Q(f)` є функція, яка для аргументу `x` повертає значення $Q(f)(x) = \operatorname{tg}(f(x))$ (тангенс від значення $f(x)$).

Формально функцію `f()` описано так, що для аргументу `x` повертається значення `sin(x)`. Але оскільки перед описом функції є декоратор `@F`, то це все одно, якби після визначення функції `f()` виконувалася команда `f=F(f)`. У результаті функція `f()` визначається як така, що значенням виразу `f(x)` є `exp(-sin(x)**2)`.

Схожа ситуація з функцією `g()`. Оскільки перед функцією є декоратор `@F`, а в тілі функції результат повертається командою `return cos(x)` (через `x` позначено аргумент функції `g()`), то насправді результатом обчислення інструкції `g(x)` є значення `exp(-cos(x)**2)`.

Щодо функції `h()`, то перед її описом є два дескриптори: `@Q` і `@F`. Це еквівалентно виконанню команди `h=Q(F(h))` після визначення функції `h()`. У підсумку значення виразу `h(x)` еквівалентне значенню виразу `tan(exp(-x**2))`.

Далі в програмному коді для декількох значень аргументу `x` (у діапазоні від 0 до 1) обчислюються значення `f(x)`, `g(x)` і `h(x)`. Для порівняння в явному вигляді обчислюються також і відповідні математичні вирази.

Аналогічно до того, як декоратори використовуються для функцій, можуть використовуватися декоратори і для класів.



Щоправда, на практиці до цього вдаються не часто, але така можливість є, і про неї варто знати.

Для створення декоратора класу необхідно мати функцію, якій аргументом передається клас і яка результатом повертає клас. На перший погляд, усе це може здатися дивним. Але тут треба згадати, що в Python

клас сам є об'єктом, а ім'я класу є посиланням на цей об'єкт. Таким чином, аргументом згаданій функції повинно передаватися ім'я класу, а в тілі функції ім'я іншого класу повинно повертатися як результат. Варіантів тут може бути багато, ми розглянемо досить простий випадок.

Отже, припустімо, що функція `F()` для аргументу — об'єкта класу `A`, повертає значення — об'єкт класу `F(A)`. Якщо клас `A` описано з декоратором `@F`, то це все одно, якби після опису класу `A` виконувалася команда `A=F(A)`. Приклад наведено в лістингу 8.6.

Лістинг 8.6. Декоратор класів

```
# Функція з аргументом - класом
# і результатом - об'єктом класу
def F(A):
    # Внутрішній клас
    class Alpha(A):
        # Метод екземпляра внутрішнього класу
        def hi(self):
            print("Клас Alpha!")
    # Результат функції - об'єкт класу
    return Alpha

# Функція з аргументом - класом
# і результатом - об'єктом класу
def Q(A):
    # Внутрішній клас
    class Bravo(A):
        # Метод екземпляра внутрішнього класу
        def hi(self):
            print("Клас Bravo!")
    # Результат функції - об'єкт класу
    return Bravo

# Клас із декоратором
@F # Декоратор класу
class First:
    # Метод екземпляра класу
    def hello(self):
        print("Клас First!")
```

```

# Клас із декоратором
@F # Декоратор класу
class Second:
    # Метод екземпляра класу
    def hello(self):
        print("Клас Second!")

# Клас із двома декораторами
@Q # Другий декоратор класу
@F # Перший декоратор класу
class Third:
    # Метод екземпляра класу
    def hello(self):
        print("Клас Third!")

# Функція для виклику методів екземпляра
def show_obj(obj):
    # Клас екземпляра
    print("Клас екземпляра:", obj.__class__)
    # Виклик методів екземпляра
    obj.hi()
    obj.hello()

# Функція для відображення характеристик класу
def show_class(A):
    # Ім'я класу
    print("Ім'я класу:", A.__name__)
    # Базовий клас
    print("Базовий клас:", A.__bases__)
    # Ланцюжок наслідування
    print("Ланцюжок наслідування:", A.__mro__)

# Створення екземплярів класів
one=First()
two=Second()
three=Third()
# Методи екземплярів
print("Екземпляри класів.")
for obj in [one, two, three]:
    show_obj(obj)

```

```
Phyton .....  
# Характеристики класів  
print("Класи.")  
for A in [First,Second,Third]:  
    show_class(A)  
.....
```

Виконання програмного коду дає такий результат:

Результат виконання програми (з лістингу 8.6)

```
Екземпляри класів.  
Клас екземпляра: <class '__main__.F.<locals>.Alpha'>  
Клас Alpha!  
Клас First!  
Клас екземпляра: <class '__main__.F.<locals>.Alpha'>  
Клас Alpha!  
Клас Second!  
Клас екземпляра: <class '__main__.Q.<locals>.Bravo'>  
Клас Bravo!  
Клас Third!  
Класи.  
Ім'я класу: Alpha  
Базовий клас: (<class '__main__.First'>,)  
Ланцюжок наслідування: (<class '__main__.F.<locals>.Alpha'>,  
<class '__main__.First'>, <class 'object'>)  
Ім'я класу: Alpha  
Базовий клас: (<class '__main__.Second'>,)  
Ланцюжок наслідування: (<class '__main__.F.<locals>.Alpha'>,  
<class '__main__.Second'>, <class 'object'>)  
Ім'я класу: Bravo  
Базовий клас: (<class '__main__.F.<locals>.Alpha'>,)  
Ланцюжок наслідування: (<class '__main__.Q.<locals>.Bravo'>,  
<class '__main__.F.<locals>.Alpha'>, <class '__main__.Third'>, <class 'object'>)
```

У цьому прикладі ми описуємо дві функції, які використовуються в декораторах класу. Також із використанням декораторів створюються три класи. Є їй інший допоміжний код. Розглянемо все це поетапно.

Аргументом функції `F()`, як ми припускаємо, передається клас (познанчений як `A`). У тілі функції описано внутрішній клас, який називається `Alpha`. Клас `Alpha` створюється на основі класу `A` шляхом наслідування. Клас `Alpha`, таким чином, наслідує атрибути класу `A`, і при цьому в тілі класу `Alpha` описано метод `hi()`. Дія методу зводиться до відображення у вікні виводу текстового значення. Посилання на внутрішній клас повертається як результат функції `F()`.

Аналогічно описано функцію `Q()`, тільки її внутрішній клас називається `Bravo`. Він також створюється наслідуванням того класу, що переданий аргументом функції `Q()`. У внутрішнього класу `Bravo` також, як і в класу `Alpha` з функції `F()`, є метод `hi()`. Цей метод також відображує повідомлення — тільки інше. Результатом функції `Q()` повертається посилання на клас `Bravo`. Тому функція `Q()`, як і функція `F()`, може використовуватися в декораторі класу.

Клас `First` описано з декоратором `@F`. У тілі класу описано метод `hello()`. Ось, що відбувається під час створення класу `First`:

- Спочатку створюється об'єкт класу `First` відповідно до того, як безпосередньо описано код класу, а посилання на об'єкт класу записується в змінну `First`. На цьому етапі клас містить лише метод екземпляра `hello()` (плюс спеціальні поля й методи).
- Далі викликається функція `F()`, аргументом якій передається посилання `First` на об'єкт відповідного класу. Під час виконання коду функції на основі класу `First` створюється внутрішній (локальний) клас `Alpha`. У цього класу, окрім методу екземпляра `hi()`, з'являється ще й метод екземпляра `hello()`.
- Посилання на об'єкт внутрішнього класу, створеного під час виклику функції `F()`, присвоюється значенням змінній `First`. У результаті, змінна посилається на об'єкт класу, в якого є методи екземпляра `hi()` (метод із класу `Alpha` функції `F()`) і `hello()` (описаний безпосередньо в класі `First`).

У цій схемі є декілька важливих для розуміння моментів. По-перше, хоча клас ми называемо `First`, насправді, змінна з такою назвою буде посылається на клас, який створювався під час виклику функції `F()`. А це локальний клас `Alpha`. Тому, якщо скористатися інструкцією `First.__name__`,

то як значення отримаємо ім'я класу Alpha. А ось базовим для цього класу є клас із іменем First. І це безпосередньо той клас, який було створено на самому початку і посилання на який відразу записувалося в змінну First. По-друге, якщо створити екземпляр для класу, на який посилається змінна First (наприклад, за допомогою команди `one=First()`), то цей екземпляр насправді буде екземпляром згаданого вище локального класу Alpha (класу, створеного під час виклику функції `F()`). У цьому легко переконатися, звернувшись до поля `__class__` екземпляра.

Клас Second також описано з декоратором `@F`. Ситуація схожа з попередньою:

- Створюється об'єкт класу Second так, як описано код цього класу. Клас містить метод екземпляра `hello()`.
- Викликається функція `F()`. Аргументом функції передається посилання на об'єкт класу Second. При цьому наслідуванням класу Second створюється внутрішній клас Alpha (але це не той клас, який створювався в класі First).
- Посилання на об'єкт створеного внутрішнього класу присвоюється змінній Second.

Таким чином, змінна Second посилається на об'єкт класу, який було створено наслідуванням «вихідного» класу Second. Якщо запитати ім'я класу Second через спеціальне поле `__name__`, то воно буде Alpha.



Хоча значення поля `__name__` для класів First і Second збігаються (йдеться про ім'я Alpha), насправді, First і Second – це різні класи. Просто значенням поля `__name__` є «локальне» ім'я класу. У цьому випадку класи створюються під час виклику функції `F()`. Кожен раз створюється новий клас із «локальним» іменем Alpha. Але оскільки йдеться про локальні простори імен під час виклику функції, конфлікту з приводу збігу назв не виникає (тобто класи з однаковими локальними назвами потрапляють у різні простори імен).

Об'єкт класу, на який посилається змінна Second, має метод екземпляра `hi()` з локального класу Alpha і метод `hello()`, описаний безпосередньо в тілі класу Second.

Складніша ситуація з класом `Third`. Цей клас описано з двома декораторами: `@F` і `@Q`. Відбувається таке:

- Створюється об'єкт класу `Third`.
- Посилання на об'єкт класу `Third` передається у функцію `F()`. У результаті створюється новий об'єкт класу: клас `Third` наслідується в локальному класі `Alpha`.
- Посилання на зазначений об'єкт передається у функцію `Q()`. У тілі цієї функції шляхом наслідування переданого аргументом класу створюється новий клас із локальною назвою `Bravo`.
- Посилання на новий об'єкт класу записується в змінну `Third`.

Отже, локальне ім'я класу, на який посилається змінна `Third`, є `Bravo`. В екземпляра цього класу є два методи: метод `hi()` із класу `Bravo`, створеного під час виклику функції `Q()`, і метод `hello()`, описаний у самому класі `Third`.



В описаній вище схемі спочатку створюється об'єкт класу з методом `hi()` з локального класу `Alpha`, а потім цей метод фактично перевизначається або перекривається методом `hi()` з локального класу `Bravo`. Втім, через екземпляр `three` і змінну `Third` можна «добрatisя» і до версії методу `hi()` з класу `Alpha`. Для цього достатньо скористатися командою `super(Third, three).hi()`, якою за допомогою функції `super()` із екземпляра `three` викликається метод `hi()`, описаний у класі, що є базовим для класу, на який посилається змінна `Third`.

Функція `show_obj()` призначена для виклику методів екземплярів класів (тих, на які посилаються змінні `First`, `Second` і `Third`). Для відображення характеристик самих класів (назва класу, базовий клас і ланцюжок наслідування) призначена функція `show_class()`.

Документування й анотації у функціях

Якщо в голові порожньо, на жаль,
найбільше відчуття гумору вас
не врятує.

Л. Керром. «Аліса в Країні Див»

Є деякі корисні властивості функцій, які можна використовувати на практиці для підвищення читабельності програмного коду. Насамперед, треба сказати про особливу роль першого (після назви функції) текстового рядка в описі функції. Цей рядок служить для документування — створення особливого коментаря, доступ до якого можна отримати програмними методами. Конкретніше, у кожної функції є спеціальне поле `__doc__` (два підкresлювання на початку, два підкresлювання в кінці). Якщо після імені функції через крапку вказати поле `__doc__`, то значенням буде текстовий рядок, розміщений на самому початку в тілі функції. Невеликий приклад наведено в лістингу 8.7.

Лістинг 8.7. Документування функції

```
# Функція
def hi():
    """Ця функція просто виводить повідомлення."""
    print("Документування функції.")
# Викликаємо функцію
hi()
# Опис функції
print(hi.__doc__)
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 8.7)

Документування функції.

Ця функція просто виводить повідомлення.

Окрім цього, в функції можна створювати «анотації» для аргументів функції. Йдеться про спеціальні коментарі, які додаються в описі аргументів функції. Також можна позначити очікуваний результат виконання функції (тип результату).

Анотації для аргументів розташовують після цих самих аргументів, як роздільник використовується двокрапка. Анотацію може бути будь-який коректний, з точки зору синтаксису Python, вираз. Якщо в аргументів є значення за замовчуванням, то вони вказуються через знак рівності після анотації до аргументу.

Анотація для результату функції вказується через стрілку `->` після круглих дужок із описом аргументів функції, але перед двокрапкою, якою закінчується рядок опису заголовка функції. Загальний шаблон функції з анотаціями для аргументів і результату такий (жирним шрифтом виділено ключові елементи):

```
def ім'я_функції(аргумент: анотація=значення,  
аргумент: анотація=значення,...)->анотація:  
    # код функції
```

Для отримання доступу до анотацій, використаних у функції, з об'єкта функції викликається спеціальне поле `__annotations__`. Значення цієї властивості — словник, ключами якого є назви аргументів, а значеннями — анотації до аргументів. Анотація до результату функції відповідає ключу `return`. Приклад створення й використання функції з анотаціями наведено в лістингу 8.8.

Лістинг 8.8. Функція з анотаціями

```
# Функція з анотаціями  
# для аргументів і результату
```

Phyton

```
def show(a:"перший аргумент",b:int=0)->None:  
    print("a =",a)  
    print("b =",b)  
# Викликаємо функцію  
show(10)  
show(10,20)  
# Використані анотації  
print(show.__annotations__)  
# Анотація для аргументу a  
annt=show.__annotations__["a"]  
print("Аргумент a:",annt)  
# Анотація для результату  
res=show.__annotations__["return"]  
print("Результат, який повертається:",res)
```

Результат виконання програмного коду такий:

■ Результат виконання програми (з лістингу 8.8)

```
a = 10  
b = 0  
a = 10  
b = 20  
{'a': 'перший аргумент', 'b': <class 'int'>, 'return': None}  
Аргумент a: перший аргумент  
Результат, який повертається: None
```

Важливо розуміти, що наявність анотацій не впливає на алгоритм виконання коду функції, так само як наявність анотації для результату функції реально не обмежує свободу програміста в плані типу значення, що повертає функція. Іншими словами, анотації — це така «декоративна» складова в описі функції. Однак деяку піктантність у визначення функції вони можуть внести. У лістингу 8.9 наведено ще один невеликий приклад функції, в якій як анотацію для аргументу і результату функції вказано команди відображення тексту у вікні виводу.

Лістинг 8.9. Анотації як команди

```
def demo(arg:print("Тут є аргумент."))->print("Результату не буде."):
    print("Вас вітає demo!")
    print(arg)
print("Викликаємо функцію demo():")
demo("Код функції виконано.")
```

Результат виконання програмного коду такий:

Результат виконання програми (з лістингу 8.9)

Тут є аргумент.
 Результату не буде.
 Викликаємо функцію demo():
 Вас вітає demo!
 Код функції виконано.

Щоб зрозуміти сенс того, що відбувається, треба врахувати, що анотації не просто «беруться до уваги» — вони виконуються. Виконуються під час створення об'єкта функції. А об'єкт функції створюється під час виконання коду з описом функції (не плутати з виконанням коду під час виклику функції). Іншими словами, анотації будуть виконані у тому місці і в той час, коли інтерпретатор «дістанеться» програмного коду з описом функції. У розглянутому прикладі команди `print("Тут є аргумент.")` і `print("Результату не буде.")`, які ми використали як анотації для аргументу і результату функції, будуть виконані під час «перегляду» інтерпретатором коду з описом функції. Це відбувається лише раз. Під час виклику функції ці команди вже виконуватися не будуть. Тому текст "Тут є аргумент." і "Результату не буде." відображується ще до того, як було викликано функцію `demo()`. А після виклику функції `demo()` ці повідомлення вже не з'являються, проте відображується текст "Вас вітає demo!" і "Код функції виконано." як наслідок виконання команд у тілі функції `demo()`.



Порядок виконання анотацій може відрізнятися від того порядку, в якому їх указано в описі функції.

Винятки як екземпляри класів

Скигленням з'їдено не повернеш.

З телесеріалу «Альф»

Із обробкою виняткових ситуацій ми трохи знайомі. Прийшов час розширити наші пізнання в цій галузі. А саме, пильнішу увагу ми приділимо тій обставині, що типи винятків описуються класами, а самі винятки є екземплярами класів. Крім того, нам треба буде познайомитися з деякими корисними механізмами — такими, наприклад, як генерування виняткових ситуацій і створення власних класів винятків.

Хоча дещо про обробку помилок (виняткових ситуацій) ми вже знаємо, тут незайвим буде нагадати основні моменти. Отже, контрольований код поміщається в блок `try`. Код, який призначений для обробки помилки, розташовується в блоці `except`. Блоків, помічених ключовим словом `except`, може бути декілька. Кожен блок відповідає помилці певного типу. Тип (або, точніше, клас) помилки, яка оброблюється у відповідному `except`-блоці, вказується після ключового слова `except`.

Для опису різних помилок передбачено спеціальні класи, які називають класами вбудованих винятків. Ці класи утворюють ієархію, яка ґрунтується на наслідуванні. Загальну ієархічну структуру класів вбудованих винятків наведено на рис. 8.1.

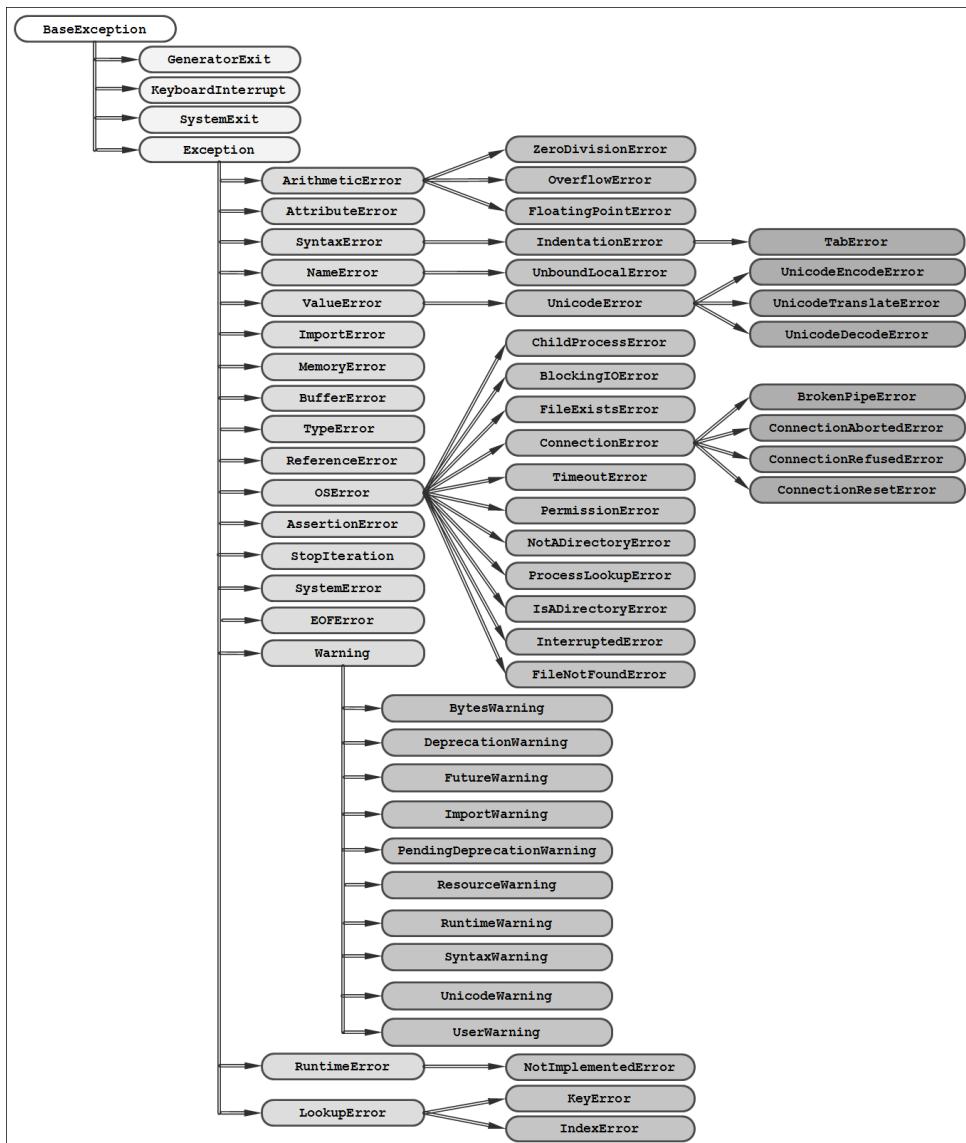


Рис. 8.1. Ієрархія класів вбудованих винятків



Як бачимо, класів досить багато. У вершині ієрархії знаходиться клас `BaseException`. У цього класу всього чотири похідних класи (`GeneratorExit`, `KeyboardInterrupt`, `SystemExit` і `Exception`), причому тільки у класу `Exception` є похідні класи. Тому без перебільшення можна сказати, що переважна більшість класів вбудованих помилок — нащадки (прямі або опосередковані) класу `Exception`.

Коли під час виконання програмного коду виникає помилка, автоматично визначається клас, якому ця помилка відповідає, і для цього класу створюється екземпляр. Образно кажучи, створений таким чином екземпляр містить основну інформацію про помилку, яка виникла. До цього екземпляру можна отримати доступ і використовувати його в явному вигляді у процесі обробки винятку.

Ми вже знаємо з попередніх розділів, як переходити винятки різних типів. Тут ми розглянемо ще декілька питань, пов'язаних із обробкою виняткових ситуацій. Практично всі вони, так або інакше, припускають використання парадигми ООП і звертання до класів вбудованих помилок. Зокрема, нас цікавитиме:

- як виконувати в одному `except`-блоці обробку для винятків декількох типів;
- як отримати доступ до екземпляра класу, що описує виняток (доступ до екземпляра винятку);
- як генерувати винятки;
- як створювати власні класи винятків.

Припустімо, необхідно реалізувати обробку виняткових ситуацій так, щоб декілька різних типів помилок оброблювалися однаково. Що треба врахувати в такому разі? Є два моменти. По-перше, якщо в `except`-блоці зазначено певний клас винятку, то переходити будуть винятки не тільки цього класу, але й винятки похідних класів. Наприклад, в `except`-блоці для винятків класу `Exception` будуть переходити практично всі винятки, оскільки клас `Exception` є базовим (за ланцюжком наслідування) для всіх класів, крім `GeneratorExit`, `KeyboardInterrupt`, `SystemExit` і `BaseException` (див. рис. 8.1). По-друге, в `except`-блоці можна зазначати не тільки один клас винятку, а цілий кортеж, елементами якого є назви класів винятків. У такому `except`-блоці будуть переходити усі винятки з кортежу. Наприклад, припустімо, що для переоплення й обробки винятку використовується такий шаблон:

```
try:  
    # контрольований код  
except (ArithmetricError, TypeError, ValueError):  
    # код для обробки  
except Warning:  
    # код для обробки
```

Що це означає? Це означає, що при виникненні помилки в `try`-блоці її буде перехоплено й оброблено в першому `except`-блоці, якщо тип помилки належить до класу `ArithmeticError` (включаючи похідні класи `FloatingPointError`, `OverflowError` і `ZeroDivisionError`), `TypeError` або `ValueError`. Якщо тип помилки інший, «у гру» вступає інший `except`-блок. У цьому блоці оброблюються помилки, які належать до класу `Warning` і всіх його похідних класів.

Ми вже знаємо, якщо є клас, то на його основі, швидше за все, можна створити екземпляр класу. Стосовно помилок ми говоримо про клас, який відповідає тій або іншій винятковій ситуації. Але де ж екземпляри таких класів? У принципі, ніхто не забороняє нам створити екземпляр класу звичайним способом, як ми до цього створювали екземпляри інших класів — через ім'я класу, зазначивши в круглих дужках (якщо треба) аргументи для конструктора й присвоївши результат такої інструкції деякій змінній (посилання на екземпляр класу). Хоча зазвичай (але не завжди) використовують ті екземпляри, що автоматично створюються під час виникнення виняткової ситуації. Тобто такий екземпляр завжди існує, якщо вже виникла помилка. Інша справа, як ми цей екземпляр використаємо (або не використаємо). У випадку, якщо в `except`-блоці ми все ж вирішили явно використати екземпляр винятку, шаблон командного коду буде таким (жирним шрифтом виділено ключові елементи коду):

```
try:
    # контролльований код
except клас_винятку as екземпляр:
    # код для обробки
```

Якщо коротко, то після назви класу винятку через ключове слово `as` зазначається формальна назва для екземпляра класу винятку, який буде автоматично створено під час виникнення помилки. Саме через це ім'я в блоці обробки винятку слід звертатися до екземпляра класу винятку. Як ілюстрацію до використання екземпляра класу винятку розглянемо невеликий приклад, наведений у лістингу 8.10.

Лістинг 8.10. Екземпляр класу винятку

```
# Імпорт функцій із модуля random
from random import seed, randint
```

Phyton

```
print("Перехоплення винятків.")  
# Список із двох елементів  
nums=[1,2]  
# Слово з шести букв  
txt="Python"  
# Цілочислова змінна  
a=10  
# Список із трьох елементів  
names=[nums,a,txt]  
# Ініціалізація генератора випадкових чисел  
seed(123)  
# Оператор циклу  
for i in range(10):  
    # Контрольований код  
    try:  
        # Випадкове ціле число в діапазоні від 0 до 2  
        n=randint(0,2)  
        print("Згенеровано число:",n)  
        # Кількість елементів у списку або тексті.  
        # При спробі викликати функцію len() для цілого  
        # числа виникає помилка (некоректний тип даних)  
        print("Кількість елементів:",len(names[n]))  
        # Неправильний індекс або ділення на нуль  
        names[n+1]//=n  
    # Обробка помилки, пов'язаної  
    # з некоректним типом даних  
    except TypeError as err:  
        # Екземпляр винятку передано  
        # як аргумент у функцію print()  
        print("Помилка:",err)  
    # Обробка помилок, пов'язаних із некоректним індексом  
    # або спробою ділення на нуль  
    except (LookupError,ArithmicError) as err:  
        print("Проблема з обчисленнями:")  
        # Визначаємо клас помилки за екземпляром класу  
        print("Клас помилки -",err.__class__)  
    # Рядок із 42 символів формується за допомогою
```

```
# оператора повтору
print("-"*42)
print("Роботу закінчено.")
```

Результат виконання програмного коду може бути таким (а може бути й іншим, оскільки багато залежить від використаного генератора випадкових чисел):

Результат виконання програми (з лістингу 8.10)

```
Перехоплення винятків.
Згенеровано число: 0
Кількість елементів: 2
Проблема з обчислennями:
Клас помилки - <class 'ZeroDivisionError'>
-----
Згенеровано число: 1
Помилка: object of type 'int' has no len()
-----
Згенеровано число: 0
Кількість елементів: 2
Проблема з обчислennями:
Клас помилки - <class 'ZeroDivisionError'>
-----
Згенеровано число: 1
Помилка: object of type 'int' has no len()
-----
Згенеровано число: 1
Помилка: object of type 'int' has no len()
-----
Згенеровано число: 0
Кількість елементів: 2
Проблема з обчислennями:
Клас помилки - <class 'ZeroDivisionError'>
-----
Згенеровано число: 0
Кількість елементів: 2
Проблема з обчислennями:
```

```
Phyton .....  
Клас помилки - <class 'ZeroDivisionError'>  
-----  
Згенеровано число: 1  
Помилка: object of type 'int' has no len()  
-----  
Згенеровано число: 2  
Кількість елементів: 6  
Проблема з обчислennями:  
Клас помилки - <class 'IndexError'>  
-----  
Згенеровано число: 2  
Кількість елементів: 6  
Проблема з обчислennями:  
Клас помилки - <class 'IndexError'>  
-----  
Роботу закінчено.
```

У цьому прикладі використовуємо генератор випадкових чисел, тому, попервах, командою `from random import seed, randint` імпортуємо з модуля `random` функції `seed()` і `randint()`. Функція `randint()` призначена для генерування цілих випадкових (псевдовипадкових) чисел. Функція `seed()` використовується для ініціалізації генератора випадкових чисел.



Хоча з генератором випадкових чисел ми вже мали справу, незайвим буде дещо пригадати.

Як би ми не намагалися згенерувати випадкове число, насправді, це непосильна задача. Максимум, чого ми можемо досягти — створити ілюзію того, що число випадкове. Тому генеровані «випадкові» числа правильніше було б називати «псевдовипадковими». Так, власне, їх і називають. Технічно процес генерування псевдовипадкових чисел полягає в тому, що за деякою формулою або алгоритмом обчислюються числа. Щоб розпочати обчислення, необхідно принаймні одне початкове, «запалювальне» число. Коли таке число задано, запускається процес обчислень. За все це відповідає група утиліт, яку ми називаємо генератором випадкових чисел. Дій, пов'язані з визначенням «початкового» стану генератора випадкових чисел, називаються ініціалізацією генератора випадкових чисел. Для виконання ініціалізації слід

указать число і передати його аргументом функції `seed()`. Оскільки, як ми вже знаємо, обчислення випадкових (псевдовипадкових) чисел — процес строго детермінований, а «початкова точка» в цьому процесі визначається числом, що використовується для ініціалізації, то вибираючи як «ініціалізатор» одне й те саме число, будемо отримувати одну й ту саму послідовність чисел. Якщо ініціалізацію генератора випадкових чисел у явному вигляді не виконано, зазвичай виконується неявна ініціалізація з використанням системного часу.

Командою `nums=[1, 2]` створюється список із двох елементів, командою `txt="Python"` створюємо текст, а числову змінну визначаємо командою `a=10`. Нарешті, командою `names=[nums, a, txt]` створюємо список із трьох елементів, один із яких — список, другий — ціле число, а третій — текст. Перший і третій елементи можна передавати аргументом функції `len()`, яка, як відомо, результатом повертає кількість елементів для списку і кількість букв для тексту.

Ініціалізація генератора випадкових чисел виконується командою `seed(123)`. У цьому випадку аргумент функції `seed()` відіграє формальну роль — можна використати й інше число.

Потім запускається оператор циклу (10 ітерацій). За кожний цикл виконується група команд, розміщених у блоці `try`. Команда `n=randint(0, 2)` генерує випадкове ціле число в діапазоні від 0 до 2 включно, і це число записується в змінну `n`. Далі в інструкції `len(names[n])` виконується спроба обчислити кількість елементів в елементі списку `names` із індексом `n`. Якщо значення змінної `n` дорівнює 0 або 2, то проблем із виконанням інструкції не виникає, оскільки елемент із індексом 0 — це список, а елемент із індексом 2 — текст. Але якщо значення змінної `n` дорівнює 1, то виникає помилка класу `TypeError`, оскільки елемент із індексом 1 — ціле число. Для перехоплення цієї помилки є спеціальний `except`-блок, у якому екземпляр помилки класу `TypeError` позначене як `err` (інструкція `TypeError as err` в `except`-блоці). У цьому `except`-блоці виконується команда `print("Помилка:", err)`. Тут екземпляр помилки `err` передано як аргумент функції `print()`, що приводить до автоматичного зведення екземпляра `err` до текстового формату (екземпляри класів винятків підтримують таку операцію). Екземпляр `err` буде замінено текстом `"object of type 'int' has no len()"` (що означає *об'єкт типу 'int' не має (функції) len()*). Усе це відбувається, нагадаємо, якщо

виконання інструкції `len(names[n])` призводить до помилки. Але якщо виконання інструкції до помилки не призводить, то помилка достоту виникне під час виконання команди `names[n+1] // = n`. Тут зроблено спробу змінити значення елемента `names[n+1]`, обчисливши результат цілочислового ділення поточного значення цього елемента на значення змінної `n`. Необхідно врахувати, що якщо справа дійшла до виконання команди `names[n+1]`, то значення `n` дорівнює 0 або 2 (при значенні 1 помилка виникає на попередньому кроці). При значенні 0 індекс `n+1` дає значення 1. Елемент із таким індексом у списку `names` є. Але біда в тому, що значення цього елемента ми намагаємося поділити на нуль. Такий самовпевнений вчинок призводить до помилки класу `ZeroDivisionError`. Якщо значення змінної `n` дорівнює 2, то ділення на нуль немає, але індекс `n+1` дорівнює 3, й елемента з таким індексом у списку `names` немає. У результаті виникає помилка класу `IndexError`. Для перехоплення і тієї, її іншої помилок призначений `except`-блок із кортежем класів винятків `LookupError` і `ArithmeticError`. Екземпляр класу помилки позначено як `err`.



Клас винятку `ZeroDivisionError` є похідним класом від класу `ArithmeticError`. Клас винятку `IndexError` є підкласом класу `LookupError`. Тому в `except`-блоці з кортежем із класів винятків `LookupError` і `ArithmeticError` перехоплюються винятки класів `ZeroDivisionError` і `IndexError`.

У блоці під час обробки помилки виконується звертання до поля `__class__` екземпляра помилки `err`. Значенням цього поля є назва класу, до якого належить екземпляр `err`.



У програмному коді є команда `print("-" * 42)`. У цьому випадку відображується рядок, який отримуємо повтором символу `"-"` 42 рази.

Як зазначалося раніше, винятки можна не тільки перехоплювати й оброблювати, а й генерувати. Навіщо генерувати винятки — питання окреме. Наприклад, генерування винятків можна розглядати як деякий спосіб імітації умовного оператора або виконання аналога безумовного

переходу в програмному коді. Але як би там не було, генерування винятків як механізм програмування існує, і ми з ним коротко познайомимося.

Із технічної точки зору, для генерування винятку (помилки) достатньо після інструкції `raise` вказати екземпляр класу винятку. Екземпляр класу винятку можна або створити, або скористатися «готовим» — наприклад, тим, що створюється автоматично, коли помилка реально виникає, і потім передається в блок обробки винятку. Приклад у листингу 8.11 ілюструє обидві ці ситуації.

Листинг 8.11. Генерування винятку

```
# Створюємо екземпляр винятку
err_one=ZeroDivisionError("Помилка ділення на нуль!")
print("Зараз \"виникне\" помилка.")
print("Перша:")
# Опис екземпляра винятку
print(err_one)
# Контрольований код
try:
    # Генерування помилки
    raise err_one
# Обробка помилки
except ZeroDivisionError as err_two:
    print("Друга:")
    # Опис помилки
    print(err_two)
    # Контрольований код
    try:
        # Повторне генерування помилки
        raise err_two
    # Обробка помилки
    except ZeroDivisionError as err_three:
        print("Третя:")
        # Опис помилки
        print(err_three)
        # Контрольований код
        try:
```

Phyton

```
# Спроба ділення на нуль
a=1/0
# Обробка помилки
except ZeroDivisionError as err_four:
    print("Четверта:")
    # Опис помилки
    print(err_four)
print("Більше жодних помилок.")
```

Виконання програмного коду дає такий результат:

■ Результат виконання програми (з лістингу 8.11)

Зараз "виникне" помилка.

Перша:

Помилка ділення на нуль!

Друга:

Помилка ділення на нуль!

Третя:

Помилка ділення на нуль!

Четверта:

division by zero

Більше жодних помилок.

У цьому програмному коді командою `err_one=ZeroDivisionError("Помилка ділення на нуль!")` ми створюємо екземпляр `err_one` винятку класу `ZeroDivisionError`. Текст, який передається як аргумент конструктору при створенні екземпляра, служить описом помилки. Саме цей текст буде результатом зведення екземпляра винятку до текстового формату: наприклад, якщо передати екземпляр `err_one` аргументом методу `print()`, в результаті у вікні виводу з'явиться текст "Помилка ділення на нуль!".

Потім у `try`-блоці виконується команда `raise err_one`, в результаті чого генерується помилка, реалізована через екземпляр `err_one` класу `ZeroDivisionError`.



Зверніть увагу, що створення екземпляра винятку не означає виникнення помилки. Для генерування помилки використовуємо інструкцію `raise`.

Для обробки згенерованої помилки призначено `except`-блок, і екземпляр оброблюваної помилки позначене в ньому як `err_two`. У самому блоці виконується команда `print(err_two)`, яка у вікні виводу відображує опис помилки. У такому разі наслідок виконання цієї команди — появу у вікні виводу повідомлення Помилка ділення на нуль!. Це те саме повідомлення, яке з'являлося під час виконання команди `print(err_one)`. Ничого дивного тут немає, оскільки змінна `err_two` посилається фактично на той самий екземпляр, на який посыпалася змінна `err_one`. Під час переходження згенерованого командою `raise err_one` винятку екземпляр винятку передається в `except`-блок, і в цьому блоці посилання на екземпляр помилки виконується через змінну `err_two`.

В `except`-блоці обробки помилки є свій `try`-блок, у якому команда `raise err_two` повторно генерує помилку. Нова обробка відбувається в `except`-блоці, і в цьому блоці посилання на екземпляр помилки виконується через змінну `err_three`. І знову йдеться про той самий екземпляр, що й у попередніх випадках. Доказом того є результат виконання команди `print(err_three)` (результат такий самий, як при виконанні команд `print(err_one)` і `print(err_two)`).

Потім при спробі виконати команду `a=1/0` знову генерується помилка, пов'язана з діленням на нуль (клас `ZeroDivisionError`). Але цього разу екземпляр винятку створюється автоматично, і це вже зовсім інший екземпляр, який до змінних `err_one`, `err_two` і `err_three` не має жодного відношення. Тому під час виконання команди `print(err_four)` в `except`-блоці (zmінна `err_four` — посилання на екземпляр винятку) отримуємо, порівняно з попередніми випадками, зовсім інший результат (текст, що описує виняток).

Для генерування винятків не обов'язково застосовувати допомогу інструкції `raise`. Можна скористатися дещо іншим підходом, в основі якого використання інструкції `assert`. Якщо після інструкції `assert` зазначити вираз із логічним значенням `False`, буде згенеровано виняток класу `AssertionError`. Невеликий приклад, у якому виняток генерується за допомогою інструкції `assert`, наведено в лістингу 8.12.

Лістинг 8.12. Використання інструкції assert

```
# Функція з логічним аргументом
def show(arg):
    # Контрольований код
    try:
        # Якщо аргумент arg дорівнює False -
        # генерується виняток
        assert arg
        # Якщо помилку не згенеровано
        print("Штатний режим.")
    # Обробка винятків
    except AssertionError as err:
        print("Виняток:",err.__class__)
# Викликаємо функцію
show(True)
show(False)
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 8.12)

```
Штатний режим.
Виняток: <class 'AssertionError'>
```

У цьому разі ми описуємо функцію `show()`, у якої, як припускається, один логічний аргумент. Цей аргумент у тілі функції зазначено після інструкції `assert`. Тому при значенні аргументу `False` генерується виняток класу `AssertionError`. Під час обробки винятку в `except`-блоці відображується значення поля `__class__` екземпляра винятку (значення поля — це клас винятку). Якщо ж аргумент функції дорівнює `True`, виняток не генерується, а виконується команда `print("Штатний режим.")` після `assert`-інструкції. Викликаючи функцію `show()` із різними аргументами, отримуємо різні результати (точніше, у вікні виведення відображується різний текст).



Під час генерування винятку за допомогою інструкції `assert` можна додати опис (текст) для екземпляра винятку. Для цього після логічного значення

В `assert`-інструкції через кому вказують текст — той текст, який буде відображення при спробі «надрукувати» екземпляр винятку. Наприклад, у результаті виконання інструкції `assert False` "Несподівана помилка" буде створено екземпляр класу `AssertionError`. Під час зведення екземпляра до текстового формату буде повернатися текстовий рядок "Несподівана помилка".

Ще одне питання, яке ми тут розглянемо, пов'язане зі створенням власних класів винятків. Загальний рецепт у цьому випадку простий: необхідно створити новий клас шляхом наслідування класу з ієрархії класів вбудованих винятків. Зазвичай як базовий клас рекомендують використовувати клас `Exception`. Процес створення власних класів винятків розглянемо на прикладі, наведеному в лістингу 8.13. У цьому прикладі ми розв'язуємо квадратні рівняння. Пошук розв'язку в числовому вигляді передбачає генерування винятків власних типів.



Нагадаємо, що квадратне рівняння має вигляд $ax^2 + bx + c = 0$. Залежно від значень параметрів a , b і c можливі такі варіанти. Формально у квадратного рівняння два розв'язки: $x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ і $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$, але це

в тому разі, якщо параметр $a \neq 0$ і вираз $D = b^2 - 4ac$ (**дискримінант** квадратного рівняння) невід'ємний, тобто якщо $D \geq 0$. Причому якщо дискримінант нульовий (тобто $D = 0$), то обидва корені рівняння збігаються:

$x_1 = x_2 = -\frac{b}{2a}$. Якщо дискримінант від'ємний (має місце $D < 0$), то на мно-

жині дійсних чисел квадратне рівняння розв'язків не має. Проте має два розв'язки на множині комплексних чисел: $x_1 = \frac{-b + i\sqrt{-D}}{2a}$ і $x_1 = \frac{-b - i\sqrt{-D}}{2a}$,

де через i позначена уявна одиниця (така, що, за визначенням, $i^2 = -1$). Усе це має місце, якщо параметр a не дорівнює нулю. Якщо ж $a = 0$, то фактично йдеться не про квадратне, а про лінійне рівняння вигляду $bx + c = 0$. У цього рівняння один розв'язок $x = -\frac{c}{b}$, але це за умови, що

$b \neq 0$. Якщо $b = 0$, то все залежить від того, чи дорівнює нулю параметр c . При $c = 0$ розв'язком рівняння $bx + c = 0$ буде будь-яке число. При $c \neq 0$ у рівняння $bx + c = 0$ розв'язків немає.

Зрозуміло, що під час написання програмного коду для розв'язку квадратних рівнянь, ураховуючи всі згадані вище фактори, можна використовувати

структуру з вкладених умовних операторів. Але ми підемо іншим шляхом. У наведеному далі програмному коді описується два класи винятків. Один виняток генерується в разі, якщо рівняння не є квадратним, а інший – якщо рівняння має комплексні розв'язки (корені).



Лістинг 8.13. Власні класи винятків

```
# Імпорт математичної функції
from math import sqrt
# Власний клас помилки
class LinearEquationWarning(Warning):
    # Конструктор
    def __init__(self,b,c):
        # Контрольований код
        try:
            # Присвоювання значення полю екземпляра.
            # Можлива помилка ділення на нуль
            self.x=-c/b
        # Якщо виникла помилка ділення на нуль
        except ZeroDivisionError:
            # Якщо параметр нульовий
            if c==0:
                # Розв'язок - будь-яке число
                self.x="будь-яке число"
            # Якщо параметр не дорівнює нулю
            else:
                # Розв'язків немає
                self.x="розв'язків немає"
    # Метод для зведення екземпляра винятку
    # до текстового формату
    def __str__(self):
        # Формується текстове значення
        # для результату методу
        txt="Розв'язок рівняння: "
        txt+=str(self.x)
        # Результат методу
        return txt
```

```

# Власний клас винятку
class ComplexRootsError(Exception):
    # Конструктор
    def __init__(self,a,b,D):
        # Значення поля екземпляра
        self.x1=complex(-b/2/a,sqrt(-D)/2/a)
        # Значення поля екземпляра
        self.x2=complex(-b/2/a,-sqrt(-D)/2/a)
    # Метод для зведення екземпляра винятку
    # до текстового формату
    def __str__(self):
        # Формування текстового рядка
        # для результату методу
        txt="x1 =" + str(self.x1) + "\n"
        txt+="x2 =" + str(self.x2)
        # Результат методу
        return txt
# Функція для відображення параметрів рівняння
def show_params(a,b,c):
    # Текст із форматуванням
    print("Параметри: a={0}, b={1}, c={2}: ".format(a,b,c))
# Функція для розв'язання квадратного рівняння
def find_roots(a,b,c):
    # Відображення параметрів розв'язуваного рівняння
    show_params(a,b,c)
    # Контрольований код
    try:
        # Якщо параметр нульовий
        if a==0:
            # Генерується виняток
            # власного типу
            raise LinearEquationWarning(b,c)
        # Дискримінант рівняння
        D=b*b-4*a*c

```

Phyton

```
# Якщо дискримінант менший від нуля
if D<0:
    # Генерується виняток
    # власного типу
    raise ComplexRootsError(a,b,D)
# Перший корінь (розв'язок) рівняння
x1=(-b+sqrt(D))/2/a
# Другий корінь (розв'язок) рівняння
x2=(-b-sqrt(D))/2/a
# Відображується корінь рівняння
print("x1 =",x1)
# Відображується корінь рівняння
print("x2 =",x2)
# Перехоплюється виняток власного типу
except LinearEquationWarning as err:
    print("Це лінійне рівняння!")
    # Відображується інформація про екземпляр винятку
    print(err)
# Перехоплюється виняток власного типу
except ComplexRootsError as err:
    print("Увага! Розв'язки комплексні!")
    # Відображується інформація про екземпляр винятку
    print(err)
# Розв'язуємо квадратні рівняння
print("Розв'язуємо рівняння a*x**2+b*x+c=0.")
# Два дійсні корені
find_roots(2,-3,1)
# Корені, що збігаються
find_roots(1,2,1)
# Два комплексні корені
find_roots(2,1,3)
# Лінійне рівняння з одним розв'язком
find_roots(0,-6,5)
# Розв'язків немає
find_roots(0,0,1)
# Розв'язок - будь-яке число
find_roots(0,0,0)
```

Результат виконання такого програмного коду наведено нижче:

■ Результат виконання програми (з лістингу 8.13)

Розв'язуємо рівняння $a*x**2+b*x+c=0$.

Параметри: $a=2$, $b=-3$, $c=1$:

$x1 = 1.0$

$x2 = 0.5$

Параметри: $a=1$, $b=2$, $c=1$:

$x1 = -1.0$

$x2 = -1.0$

Параметри: $a=2$, $b=1$, $c=3$:

Увага! Розв'язки комплексні!

$x1 = (-0.25+1.1989578808281798j)$

$x2 = (-0.25-1.1989578808281798j)$

Параметри: $a=0$, $b=-6$, $c=5$:

Це лінійне рівняння!

Розв'язок рівняння: 0.8333333333333334

Параметри: $a=0$, $b=0$, $c=1$:

Це лінійне рівняння!

Розв'язок рівняння: розв'язків немає

Параметри: $a=0$, $b=0$, $c=0$:

Це лінійне рівняння!

Розв'язок рівняння: будь-яке число

Оскільки в процесі обчислень ми збираємося добувати квадратний корінь, командою `from math import sqrt` із модуля `math` імпортуємо функцію добування квадратного кореня `sqrt()`.

Клас із назвою `LinearEquationWarning` створюється шляхом наслідування класу винятку `Warning`. У тілі класу описано конструктор, і серед аргументів цього конструктора, крім посилання `self` на екземпляр класу, є ще параметри `b` і `c`. Назви цих аргументів збігаються з назвами параметрів розв'язуваного рівняння.



Параметр a в цьому випадку не потрібен, оскільки помилка класу `LinearEquationWarning` генеруватиметься в разі, якщо параметр a дорівнює нулю. Тобто, якщо вже справа дійшла до створення екземпляра винятку `LinearEquationWarning`, то це автоматично означає, що параметр a у рівнянні нульовий.

У тілі конструктора командою `self.x=-c/b` полю x екземпляра класу присвоюється значення, що дорівнює, очевидно, кореню рівняння (лінійного в цьому випадку). Але тут можлива помилка ділення на нуль — якщо параметр b нульовий. У такому разі на сцену виходить обробник винятку класу `ZeroDivisionError` і за допомогою умовного оператора, залежно від значення параметра c , полю x екземпляра класу присвоюється значення "будь-яке число" або "розв'язків немає".

Також у класі `LinearEquationWarning` описано метод `__str__()`, що дозволяє автоматично зводити екземпляри класу до текстового формату. У тілі методу формується текстовий рядок, що містить, крім іншого, значення поля x екземпляра класу. Цей текст повертається як результат методу.

Ще один клас винятку називається `ComplexRootsError` і створюється він наслідуванням класу `Exception`. Конструктору, описаному в тілі класу, передаються параметри рівняння a і b , а також дискримінант рівняння D . Неявно припускається, що дискримінант від'ємний (бо тільки в цьому випадку генерується помилка класу `ComplexRootsError`). У тілі конструктора полям $x1$ і $x2$ екземпляра класу присвоюються значення. Це — комплексні числа. Комплексні числа створюються за допомогою вбудованої функції `complex()` (нагадаємо, що її аргументи — це дійсна й уявна частини комплексного числа). Під час обчислень використано формули для комплексних коренів квадратного рівняння (див. вставку вище).

Під час зведення екземпляра класу до текстового формату (програмний код методу `__str__()`) у текстовий рядок, що повертається резултатом, добавляються значення полів $x1$ і $x2$ екземпляра класу.

Також у програмному коді, крім опису власних класів винятків, є функція `show_params()`, призначена для відображення значень параметрів для розв'язуваного квадратного рівняння.



У тілі функції `show_params()` виконується команда `print("Параметри: a={0}, b={1}, c={2}: ".format(a,b,c))`. Тут ми вдалися до формування текстового рядка для відображення у вікні виводу за допомогою виклику функції `format()` із рядка формату. У тексті "Параметри: a={0}, b={1}, c={2}:", із якого викликається функція `format()`, інструкції `{0}`, `{1}` і `{2}` визначають місце вставки значень відповідно першого, другого й третього аргументів функції `format()`.

Функція `show_params()` викликається в тілі іншої функції, яка називається `find_roots()` і призначена для розв'язання рівняння. Крім цього, у тілі функції `find_roots()` в умовному операторі перевіряється умова `a==0`, і якщо воно істинна, команда `raise LinearEquationWarning(b, c)` генерує виняток класу `LinearEquationWarning`.



Командою `LinearEquationWarning(b, c)` створюється екземпляр класу `LinearEquationWarning`, але посилання на цей екземпляр явно в жодну змінну не записується. Це так званий **анонімний** екземпляр класу.

Якщо помилка не генерується (умова `a==0` хибна), то командою `D=b*b-4*a*c` обчислюється дискримінант рівняння. У разі, якщо він від'ємний (умова `D<0` ще в одному умовному операторі), командою `raise ComplexRootsError(a, b, D)` генерується помилка користувачького класу `ComplexRootsError`.



Команда `ComplexRootsError(a, b, D)` створює анонімний екземпляр класу `ComplexRootsError`.

Якщо умова `D<0` не виконується, командами `x1=(-b+sqrt(D))/2/a` і `x2=(-b-sqrt(D))/2/a` обчислюються корені рівняння (при цьому,

якщо дискримінант нульовий, то значення коренів однакові), а потім командами `print("x1 =", x1)` і `print("x2 =", x2)` обчислені значення відображуються у вікні виводу.

Під час перехоплення й обробки винятку типу `LinearEquationWarning` командою `print("Це лінійне рівняння!")` виводиться текстове повідомлення, а потім командою `print(err)` відображується й опис екземпляра винятку `err`. Аналогічно відбувається обробка винятку класу `ComplexRootsError`.



У принципі, існує можливість перехоплювати й оброблювати винятки класів `LinearEquationWarning` і `ComplexRootsError` в одному `except`-блоці. При цьому «специфічні» для кожного з використаних нами `except`-блоків текстові повідомлення можна було б «сховати» у конструкто-ри екземплярів відповідних класів.

Те, що класи `LinearEquationWarning` і `ComplexRootsError` створювалися наслідуванням різних базових класів (`Warning` і `Exception` відповідно) у контексті їхнього використання не є принциповим. Але відмінність все ж таки існує. Наприклад, у `except`-блоці для перехоплення винятків класу `Exception` будуть перехоплюватися й помилки класів `LinearEquationWarning` і `ComplexRootsError`, а в `except`-блоці для перехоплення винятків класу `Warning` перехоплюватимуться помилки тільки класу `LinearEquationWarning`.

Для ілюстрації того, як працює вся ця схема, у програмному коді наведено декілька прикладів виклику функції `find_roots()` із різними наборами параметрів для розв'язуваного рівняння.

Ітератори і функції-генератори

Я терпляча людина, у що мені важко повірити.

Дж. Буш (молодший)

У Python є особлива категорія об'єктів, які називають *ітераторами*. Ітератори створюються на основі *ітераційних об'єктів*. До ітераційних об'єктів відносять списки, кортежі й текст. Але нас цікавить створення власних ітераційних об'єктів. Такі об'єкти створюються на основі класів. Класи, на основі яких створюються ітераційні об'єкти, називатимемо *ітераційними класами* або *класами-ітераторами* (хоча остання назва, можливо, й не дуже коректна). Таким чином, усе починається з класу-ітератора. Якщо підійти до питання формально, то клас-ітератор — це клас, у якому підтримуються (тобто, визначені) спеціальні методи `__iter__()` і `__next__()`. Але щоб зрозуміти, навіщо все це потрібно і як використовується, краще «почати здалеку». І насамперед, розглянемо програмний код, наведений у лістингу 8.14.

Лістинг 8.14. Функції `next()` і `iter()`

```
s=iter([1,2,3])
print(next(s))
print(next(s))
print(next(s))
try:
    print(next(s))
except Exception as err:
    print(err.__class__)
```

Результат виконання програмного коду такий:

 Результат виконання програми (з лістингу 8.14)

```
1  
2  
3  
<class 'StopIteration'>
```

Ми виконуємо прості операції, в яких використовуємо функції `iter()` і `next()`. А саме, змінній `s` присвоюється результат виразу `iter([1, 2, 3])`, у якому як аргумент функції `iter()` передається список `[1, 2, 3]`. Одразу зазначимо, що результатом виразу `iter([1, 2, 3])` є *ітератор* — цей об'єкт можна передавати функції `next()`. Причому кожного разу під час виклику функції `next()` (із аргументом-ітератором) отримуємо нове значення — щоправда, до певного моменту. Свідченням того є результат послідовного виконання команди `print(next(s))`: під час перших трьох викликів повертаються відповідно числові значення 1, 2 і 3. Нескладно здогадатися, що це елементи списку `[1, 2, 3]`, який передається як аргумент функції `iter()`. А ось під час виконання команди `print(next(s))` у четвертий раз, генерується помилка класу `StopIteration`. Це, в принципі, теж зрозуміло — у списку `[1, 2, 3]` за-кінчилися елементи. Що з цього випливає? А випливає буквально ось що: ітератор — це такий об'єкт, який можна передавати як аргумент функції `next()` і отримувати в результаті деяке значення (насправді, не обов'язково числове). Триває це не до нескінченності. У якийсь момент під час виклику функції `next()` із ітератором як аргументом генерується виняток класу `StopIteration`. Виникає питання: а яка в усьому цьому роль функції `iter()`? Відповідь проста: за допомогою функції `iter()` на основі ітераційного об'єкта створюється ітератор. У розглянутому прикладі в нас був список `[1, 2, 3]`. Список — це ітераційний об'єкт, але ще не ітератор. Ітератор на основі списку треба створити. Щоб створити ітератор на основі списку, список передається аргументом функції `iter()`.



Ми досить часто використовували й використовуємо інструкції з ключовим словом `for` виду `for` елемент `in` щось. Обробляються вони насправді так. Спочатку створюється ітератор `iter(щось)`. Потім цей ітератор послідовно під час кожної ітерації передається функції `next()`. Ітерації

тривають доти, поки під час виклику функції `next()` не буде згенеровано виняток `StopIteration`.

Але функції `next()` і `iter()` — це, так би мовити, зовнішня частина айсберга. Розглянемо програмний код, поданий у лістингу 8.15.

Лістинг 8.15. Методи `__next__()` і `__iter__()`

```
s=[1,2,3].__iter__()
print(s.__next__())
print(s.__next__())
print(s.__next__())
try:
    print(s.__next__())
except Exception as err:
    print(err.__class__)
```

Результат виконання програмного коду буде таким самим, як у попередньому випадку:

Результат виконання програми (з лістингу 8.15)

```
1
2
3
<class 'StopIteration'>
```

Ситуація дуже нагадує розглянутий раніше приклад. Тільки тепер замість функції `iter()` із аргументом-списком зі списку викликається метод `__iter__()`, а замість функції `next()` із аргументом-ітератором із ітератора викликається метод `__next__()`. Нескладно здогадатися, що виклик функцій `iter()` і `next()` приводить до виклику відповідно методів `__iter__()` і `__next__()`. Далі логіка така: нам треба було б описати клас із методом `__iter__()`. Екземпляри такого класу — ітераційні об'єкти. Під час виклику з ітераційного об'єкта методу `__iter__()` як результат повинен повернатися ітератор — об'єкт, у якого є метод `__next__()`. Такий об'єкт теж потрібно створювати на основі якось класу (точніше такого, в якому описано метод `__next__()`). Природно спадає на думку рішення: описати один клас із методами `__iter__()`

`i __next__()`, причому метод `__iter__()` визначити так, щоб як результат він повертає посилання на екземпляр, із якого викликається. У підсумку, клас потрібен всього один, а екземпляр, який створюється на основі цього класу, підтримує обидва методи `__iter__()` і `__next__()`. Тобто отримуємо наочне ітераційний об'єкт й ітератор «в одній особі». Щоб заробити такий подвійний приз, нам потрібен усього один клас. Саме про це й ішлося на початку пункту, коли ми говорили про класи-ітератори. Далі для простоти, і якщо це не буде призводити до непорозумінь, екземпляри класів-ітераторів будемо називати ітераторами.

Отже, схема наших дій така:

- Створюємо клас із методами `__iter__()` і `__next__()`.
- Метод `__iter__()` описуємо так, щоб як результат він повертає посилання на екземпляр, із якого викликається.
- «Правильний» метод `__next__()` повинен бути таким: якусь кількість викликів методу повертає значення, а потім генерує виняток класу `StopIteration`.

У лістингу 8.16 наведено приклад створення ітератора, який дозволяє обчислювати числа з послідовності Фібоначчі.



Нагадаємо, що в послідовності Фібоначчі перші два числа дорівнюють 1, а кожне наступне число дорівнює сумі двох попередніх: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 і так далі.

Лістинг 8.16. Ітератор для чисел Фібоначчі

```
# Клас - ітератор
class Fibs:
    # Конструктор
    def __init__(self, n):
        # Початкові значення для полів
        # count (номер генерованого числа),
        # a (генероване число)
        # i b (наступне число)
        self.start()
        # Значення поля n (кількість генерованих
```

```

    # чисел)
    self.n=n
# Метод для переведення в "початковий стан"
# полів count, a і b
def start(self):
    self.count=1 # Номер генерованого числа
    self.a=1     # Генероване число
    self.b=1     # Наступне число
# Метод, який повертає ітератор
def __iter__(self):
    # Результат методу - посилання
    # на екземпляр класу
    return self
# Метод для обчислення числа Фіbonаччи
def __next__(self):
    # Якщо перевищено ліміт генерованих чисел
    if self.count>self.n:
        # Поля count, a і b отримують початкові
        # одиничні значення
        self.start()
        # Генерується виняток
        # класу StopIteration
        raise StopIteration
    # Якщо ліміт генерування
    # чисел не перевищено
    res=self.a # Значення, що повертається методом
    # Нове число в послідовності Фіbonаччи
    self.a,self.b=self.b,self.a+self.b
    # Нове значення поля count
    self.count+=1
    # Результат методу
    return res
# Створюється екземпляр класу
obj=Fibs(10)
# Екземпляр класу використовується в операторі циклу
for s in obj:
    print(s,end=" ")

```

Phyton

```
print()
# Змінюємо значення полів екземпляра класу
obj.a=3      # Четверте число в послідовності
obj.b=5      # П'яте число в послідовності
obj.count=4 # Номер числа в послідовності
# Новий оператор циклу з екземпляром класу
for s in obj:
    print(s,end=" ")
print()
# Кількість генерованих чисел
obj.n=15
# Ще один оператор циклу з екземпляром класу
for s in obj:
    print(s,end=" ")
```

Результат виконання цього програмного коду такий:

Результат виконання програми (з лістингу 8.16)

```
1 1 2 3 5 8 13 21 34 55
3 5 8 13 21 34 55
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Клас, який ми створюємо в програмному коді і який призначений для створення ітераторів, називається Fibs. В екземпляра класу є такі поля:

- Числове поле `n` визначає кількість чисел у послідовності Фібоначчі, які генеруватимуться ітератором (він же — екземпляр класу).
- Числове поле `count` визначає номер того елемента послідовності, який генеруватиметься під час виклику функції `next()`. Очевидно, що генерування чисел може здійснюватися, якщо значення поля `count` не перевищує значення поля `n`. В іншому разі генеруватиметься виняток класу `StopIteration`.
- Поля `a` і `b` «пам'ятають» два числа з послідовності Фібоначчі: у поле `a` записується значення числа, яке буде згенеровано під час виклику функції `next()`, а в поле `b` записується наступне число в послідовності.

Метод `start()` екземпляра класу потрібен для встановлення початкових значень полів `a`, `b` і `count`. У тілі методу всі ці поля отримують одиничні значення.

У конструкторі викликається метод `start()`, а також присвоюється значення полю `n`. Значення для цього поля передається як аргумент конструктору.

Метод `__iter__()` повинен, як уже зазначалося, повернати посилання на ітератор. Також ми домовилися, що ітератором буде екземпляр класу `Fibs`. Тому в тілі методу `__iter__()` із аргументом `self` усього одна інструкція `return self`, якою результатом методу повертається посилання на екземпляр класу `Fibs`, із якого викликається метод (і цей екземпляр, відповідно, передається як аргумент функції `iter()`).

Найцікавіший код, мабуть, у методу `__next__()`. Саме в цьому методі визначається, яке число буде генеруватися ітератором (екземпляром класу) на кожній ітерації. У тілі методу, насамперед, в умовному операторі, перевіряється умова `self.count > self.n` (через `self` позначено посилання на екземпляр класу — аргумент методу `__next__()`). Істинність цієї умови означає, що числа більше генеруватися не повинні. У цьому випадку командою `self.start()` полям `a`, `b` і `count` екземпляра `self` присвоюються одиничні значення (як на початку ітераційного процесу), а потім командою `raise StopIteration` генерується виняток класу `StopIteration`.



Зверніть увагу, що виняток генерується, але не перехоплюється і не оброблюється. Його буде оброблено оператором `for`, і саме для цього оператор призначено. Поява винятку є ознакою того, що ітераційний процес необхідно закінчити.

Оскільки виняток не перехоплюється, то `try`-блок ми не використовуємо. Також після ключового слова `raise` ми вказали тільки ім'я класу винятку (а не інструкцію створення екземпляра, нехай навіть анонімного). Так можна робити. У такому разі створюється анонімний екземпляр винятку відповідного класу.

Якщо умову в умовному операторі не виконано, метод повинен повернути результатом чергове число з послідовності Фібоначчі. У цьому випадку виняток не генерується, проте виконується послідовність таких команд:

- Командою `res=self.a` у локальну змінну `res` записується значення поля `a`. Саме це число буде повертатися результатом методу.
- Командою `self.a, self.b=self.b, self.a+self.b` обчислюється нове число в послідовності Фібоначчі, і одночасно присвоюються нові значення полям `a` і `b`. Тут використовується *множинне присвоювання*: спочатку за старими значеннями полів `a` і `b` обчислюються вирази в правій частині, а потім ці вирази присвоюються відповідно полю `a` і `b`.
- Командою `self.count+=1` значення поля `count` збільшується на одиницю (оскільки генерується нове число).
- Нарешті, командою `return res` значення змінної `res` повертається як результат методу.

На цьому опис класу `Fibs` закінчується. Далі йдуть команди, які ілюструють використання екземплярів цього класу як ітераторів. Так, командою `obj=Fibs(10)` створюється екземпляр `obj` класу `Fibs`. Аргумент `10`, переданий конструктору, означає, що екземпляр `obj` дозволить генерувати `10` перших чисел у послідовності Фібоначчі. Щоб перевірити це, запускається оператор циклу, в якому змінна `s` перебирає значення «з екземпляра» `obj`. За кожний цикл виконується команда `print(s, end=" ")`, унаслідок чого числа (значення змінної `s`) виводяться через пробіл в одному рядку. У результаті отримуємо послідовність Фібоначчі з `10` чисел. Якщо після цього запустити ще раз оператор циклу, отримуємо такий самий результат. Тобто екземпляр `obj` є «багаторазовим» ітератором — у тому сенсі, що його можна використовувати багато разів.



У цьому полягає важливий «ідеологічний» момент. Річ у тім, що кожен раз під час виклику методу `__next__()` змінюється «внутрішній стан» екземпляра, з якого викликається метод: змінюються значення полів екземпляра. Теоретично після того, як під час чергового виклику методу `__next__()` буде згенеровано виняток класу `StopIteration` й ітераційний процес закінчиться, значення полів екземпляра будуть зовсім не такими, як на початку ітераційного процесу. Тому, щоб екземпляр можна було використовувати знову, необхідно повернути його «внутрішні налаштування» у «початковий

стан». Саме з цією метою в тілі методу `__next__()` перед генеруванням винятку викликається метод `start()`. Якщо цього не зробити, то отримаємо «одноразовий» ітератор: такий екземпляр можна буде використовувати в операторі циклу один раз. Під час спроби використовувати його наступний раз жодних ітерацій не буде — виняток `StopIteration` згенерується під час першого ж виклику методу `__next__()`.

Можна було вчинити й інакше: помістити команду виклику методу `start()` не в тілі методу `__next__()`, а в тілі методу `__iter__()`. Хто бажає, може подумати, що змінилося б у цьому випадку.

Щоб докладніше розкрити механізм генерування чисел за допомогою екземпляра `obj`, далі проводимо невеликий експеримент. Командами `obj.a=3`, `obj.b=5` і `obj.count=4` змінюємо значення полів екземпляра `obj`. У такому разі поле `a` дорівнює четвертому (по порядку) числу в послідовності Фібоначчі, наступне число в послідовності — це значення поля `b`, і значення поля `count` — відповідає порядковому номеру числа, записаного в поле `a`. Якщо тепер запустити оператор циклу, то генерування чисел почнеться з числа з порядковим номером 4 у послідовності. Останнє згенероване число — десяте (відповідно до значення поля `n` екземпляра `obj`, а воно дорівнює 10).



Після закінчення оператора циклу поля `a`, `b` і `count` знову матимуть одиничні значення.

Якщо змінити значення поля `n` екземпляра `obj`, наприклад, командою `obj.n=15`, то під час чергового запуску оператора циклу буде генеруватися не 10, а вже 15 чисел із послідовності Фібоначчі.

У певному сенсі *функція-генератор* може розглядатися як «спрощена» і «полегшена» форма для створення ітератора або ітераційного об'єкта (теж «в одній особі»). Як і ітератор, результат виклику функції-генератора під час кожного чергового звертання до нього дає нове значення з певного набору значень. Для простоти будемо називати результат виклику функції-генератора *об'єктом генератора*. Об'єкт генератора використовується за тією ж схемою, що й ітератор.

Під час створення функції-генератора не треба явно описувати методи `__next__()` і `__iter__()` (вони створюються автоматично). Немає необхідності явно створювати об'єкт генератора (результат функції). Усе, що треба зробити, — це «сформувати» послідовність значень, які будуть повертатися як об'єкт генератора. Головна формальна відмінність функції-генератора від звичайної функції полягає в тому, що результат функції-генератора формується за допомогою ключового слова `yield` (а не повертається інструкцією `return`, як у звичайних функцій). Приклад функції-генератора, призначеної для генерування чисел із послідовності Фібоначчі, подано в лістингу 8.17.

Лістинг 8.17. Функція-генератор для чисел Фібоначчі

```
# Функція-генератор для чисел Фібоначчі
def fibs_gen(n):
    # Перше число послідовності
    a=1
    # Друге число послідовності
    b=1
    # Оператор циклу для обчислення
    # чисел Фібоначчі
    for i in range(n):
        # Значення для результата
        # функції (для цієї ітерації)
        res=a
        # Обчислення нового значення
        # в послідовності й присвоювання
        # значень змінним a і b
        a,b=b,a+b
        # Результат функції-генератора
        yield res
print("Спроба №1")
# Оператор циклу для виводу 10 чисел
# із послідовності Фібоначчі
for s in fibs_gen(10):
    print(s,end=" ")
print("\nСпроба №2")
# Ще один оператор циклу для виводу
```

```
# 10 чисел із послідовності Фібоначчі
for s in fibs_gen(10):
    print(s,end=" ")
print("\nСпроба №3")
# Посилання на об'єкт генератора записано
# в змінну
f=fibs_gen(15)
# Оператор циклу для виводу 15 чисел
# із послідовності Фібоначчі.
# Посилання на об'єкт генератора виконано
# через змінну
for s in f:
    print(s,end=" ")
print("\nСпроба №4")
# Ще один оператор циклу для виводу
# 15 чисел із послідовності Фібоначчі.
# Посилання на об'єкт генератора виконано
# через змінну
for s in f:
    print(s,end=" ")
print("Закінчення роботи.")
```

Результат виконання програмного коду такий:



Результат виконання програми (з листингу 8.17)

```
Спроба №1
1 1 2 3 5 8 13 21 34 55
Спроба №2
1 1 2 3 5 8 13 21 34 55
Спроба №3
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Спроба №4
Закінчення роботи.
```

У цьому прикладі, як зазначалося вище, ми створюємо функцію-генератор для чисел Фібоначчі. Функція називається `fibs_gen()`, і в неї один аргумент (позначене як `n`) — кількість генерованих функцією чисел. У тілі

функції змінним `a` і `b` присвоюються одиничні значення. Це — перші два числа послідовності. Потім запускається оператор циклу, в якому змінна `i` «пробігає» значення від 0 до `n-1`. Конкретні межі діапазону зміни змінної `i` не важливі, оскільки в тілі циклу ця змінна явно не використовується. Головне, що цикл складається з `n` ітерацій. За кожну ітерацію командою `res=a` у змінну `res` записується поточне значення змінної `a`. Це значення буде «повернено» (на цій ітерації) як результат функції. Потім командою `a,b=b,a+b` із використанням множинного присвоювання обчислюється нове число в послідовності Фіbonacci, і змінні `a` і `b` отримують нові значення.



Вираз `a,b=b,a+b` розраховується так. Для поточних значень змінних `a` і `b` обчислюються значення виразів `b` (позначимо як `значення_1`) і `a+b` (позначимо як `значення_2`) у правій частині від оператора присвоювання. Потім змінній `a` присвоюється обчислене на попередньому етапі `значення_1`, а змінній `b` присвоюється обчислене на попередньому етапі `значення_2`.

Після цього командою `yield res` формується результат (для цієї ітерації).



Щодо результату функції-генератора, то доцільніше на все це дивитися, як на послідовність значень, які повертаються у процесі виконання програмного коду функції. Команда з інструкцією `yield` дає або додає новий елемент у цю послідовність. Нерідко для формування такої послідовності результатів звертаються за допомогою до оператора циклу. Однак це не обов'язково. Наприклад, можемо визначити таку функцію-генератор:

```
def colors():
    yield "Червоний"
    yield "Жовтий"
    yield "Зелений"
```

Тоді під час виконання оператора циклу

```
for clr in colors():
    print(clr)
```

отримуємо такий результат:

Червоний
Жовтий
Зелений

Для перевірки роботи функції запускаємо оператор циклу, в якому змінна `s` «пробігає» значення з об'єкта генератора, який обчислюється інструкцією `fibs_gen(10)` (викликаємо функцію `fibs_gen()` із аргументом `10`). У тілі оператора циклу виконується команда `print(s, end=" ")`. Як наслідок, у вікні виводу в рядок через пробіл виводиться 10 чисел із послідовності Фібоначчі.

Що відбувається під час виклику функції-генератора `fibs_gen()`? Під час виклику функції-генератора створюється об'єкт генератора. Для об'єкта генератора можна викликати функції `iter()` і `next()` (що неявно й відбувається під час використання функції-генератора в операторі циклу). Однак тут є важлива обставина: створений у результаті виклику функції-генератора об'єкт генератора «одноразовий». Його можна використовувати в операторі циклу тільки один раз. На перший погляд, це може здатися дивним. Особливо, якщо врахувати, що повторно виконаний оператор циклу в нашому прикладі дає такий самий результат, як і перший оператор циклу. Але все стає на свої місця, якщо врахувати, що кожний раз під час виклику функції-генератора (навіть якщо з тими ж аргументами) створюється новий об'єкт. Він і використовується. Щоб переконатися в справедливості цього твердження, виконаємо просту перевірку. Спочатку командою `f=fibs_gen(15)` у змінну `f` записуємо посилення на об'єкт генератора, створений викликом функції-генератора `fibs_gen(15)`. Потім в операторі циклу вказуємо зміну `f`. Спочатку все відбувається цілком очікувано: у вікні виводу відображується 15 чисел із послідовності Фібоначчі. Але якщо ми спробуємо виконати достоту та-кий же оператор із тією самою змінною `f`, жодне число виведено не буде. Причина в тому, що змінна `f` посилається на той самий об'єкт, що використовувався в першому (із двох останніх) операторі циклу. І цей об'єкт свою задачу виконав. Більше від нього користі немає.



Класи-ітератори й функції-генератори можна використовувати не тільки для створення об'єктів, які служать «контейнерами» значень, що перебираються в операторах циклу. На основі ітераторів можна створювати, наприклад, списки. Для цього достатньо ітератор передати аргументом функції

`list()`. Скажімо, результатом інструкції `list(fibs_gen(15))` буде список із 15 чисел із послідовності Фібоначчі (функцію `fibs_gen()` описано в лістингу 8.17). До аналогічного результату приведе виконання інструкції `list(Fibs(15))` (клас `Fibs` описано в лістингу 8.16).

Крім того, є деякі вбудовані функції, що дозволяють просто й швидко створювати об'єкти ітераційного типу. Наприклад, функція `map()`. Якщо першим аргументом функції `map()` передати ім'я деякої функції, а другим аргументом — список значень (команда формату `map(функція, список)`), то в результаті отримуємо об'єкт ітераційного типу. Кожне значення, що повертається під час звертання до цього об'єкта за допомогою функції `next()`, отримується викликом функції, вказаної першим аргументом у `map()`, з аргументом, який є елементом списку, переданого другим аргументом функції `map()`. Якщо використовувати інструкцію вигляду `list(map(функція, список))`, у результаті матимемо список, елементи якого отримані викликом функції з аргументом, який «пробігає» значення елементів списку. Наприклад, результатом виразу `list(map(lambda n: 2**n, [1, 2, 3]))` буде список `[2, 4, 8]`.

За схожим принципом діє функція `filter()`. Результатом виразу вигляду `list(filter(функція, список))` є список, який отримано зі списку, переданого аргументом функції `filter()` — але тільки тих елементів, для яких виклик функції (перший аргумент функції `filter()`) повертає значення `True`. Тобто фактично в цьому випадку можна виконувати «фільтрацію» списків. Ім'я функції, що відіграє роль критерію включення елемента у список-результат, зазначається першим аргументом функції `filter()`. Наприклад, результатом виразу `list(filter(lambda x: bool(x%2), [1, 5, 6, 2, 7, 8]))` буде список `[1, 5, 7]` (тільки непарні елементи).

Функція `zip()` дозволяє з декількох списків створити список кортежів, де кожний кортеж отримують включенням відповідних елементів із вихідних списків. Наприклад, якщо `x=[1, 2, 3], y=[4, 5, 6] i z=[7, 8, 9]`, то результатом виразу `list(zip(x, y, z))` буде список `[(1, 4, 7), (2, 5, 8), (3, 6, 9)]`.

Варто також мати на увазі, що створювані в результаті виклику функцій `map()`, `filter()` і `zip()` об'єкти ітераційного типу є «одноразовими» в тому контексті, як це описувалося вище. Так, якщо виконати команду `obj=map(lambda n: 2**n, [1, 2, 3])`, а потім інструкцію `list(obj)`, то цілком очікувано отримаємо в результаті список `[2, 4, 8]`. Але, якщо на наступному етапі використовувати об'єкт `obj` в операторі циклу, ітерації не буде! Хоча, якщо перед оператором циклу інструкцію `list(obj)` не застосовувати, то оператор циклу виконується. Причина в тому, що об'єкт `obj` може використовуватися тільки один раз. Ми з такою ситуацією вже зустрічалися.

Резюме

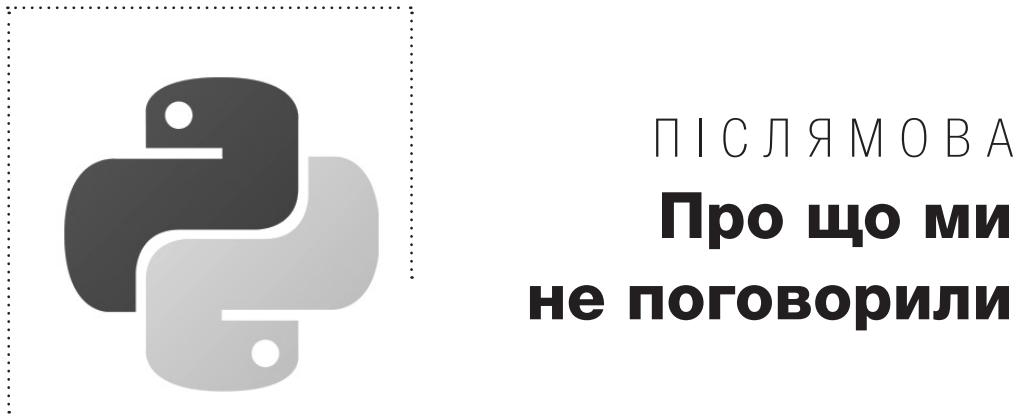
Це ваші гроші, ви заплатили за них.

Дж. Буш (молодший)

- У Python можна створювати функції зі змінною (нефіксованою) кількістю аргументів. У весь набір таких аргументів обробляється як список (кортеж), а при описі функції позначається як один аргумент, але із зірочкою * перед назвою аргументу.
- Аргумент, помічений подвійною зірочкою **, позначає словник із іменованими аргументами (переданими функції за ключем).
- Декоратори дозволяють змінювати поведінку та властивості функцій і класів. Декоратор для функцій створюється на основі функції, якій аргументом передається функція і яка повертає результатом функцію. Декоратор класу створюється на основі функції, аргументом якої є клас і результатом теж є клас. Декоратор (символ @ ім'я функції для створення декоратора) вказується перед іменем функції або класу. У результаті функція декоратора застосовується до «декорованої» функції або класу. Результат застосування декоратора до функції або класу записується у змінну, яка служить назвою відповідно функції або класу.
- Перший текстовий рядок в описі функції слугує як текст довідки по функції та повертається як значення поля __doc__ об'єкта функції.
- Для аргументів і результату функції можна створювати анотації: спеціальні текстові «пояснення». Анотація для аргументів зазначається через двокрапку після імені аргументів в описі

функції. Анотація для результату вказується після круглих дужок (у яких описуються аргументи функції), через інструкцію `->`. Наявність анотацій не накладає обмежень на тип аргументів або результату функції.

- Класи вбудованих винятків утворюють ієрархію, на вершині якої знаходиться клас `BaseException`. Інший важливий клас — клас `Exception`. Більшість класів винятків є похідними класами від класу `Exception`.
- В `except`-блоці перехоплюються не тільки ті винятки, які вказано явно, але ще й винятки похідних класів.
- Під час обробки винятку в `except`-блоці може використовуватися посилання на екземпляр винятку.
- Виняток може бути згенеровано вручну за допомогою інструкції `raise` ї `assert`.
- Власні класи винятків створюються наслідуванням класів вбудованих винятків (зазвичай базовим є клас `Exception`).
- Ітератори створюються на основі класів, що містять опис методів `__iter__()` і `__next__()`. Метод `__iter__()` повертає результатом посилання на екземпляр, із якого він був викликаний. Під час виклику методу `__next__()` кожен раз повертається деяке значення, поки під час чергового виклику не буде згенеровано виняток класу `StopIteration`.
- Ітератор можна використовувати в операторі циклу (як «контейнер» для перебирання значень у циклі) або, наприклад, для формування списків.
- Функція-генератор повертає об'єкт, який може використовуватися як ітератор. У тілі функції-генератора результат формується за допомогою інструкції `yield`.



ПІСЛЯМОВА
**Про що ми
не поговорили**

Ганьба! Слава! Тобто добрий вечір!

«Шоу Довгоносиків»

Зазвичай, про програмування говорять і пишуть сухою технічною мовою. У принципі, це правильно. Але щодо мови Python, то «ліричні» епітети приходять самі собою, так би мовити. Їх багато. Якщо спробувати охарактеризувати цю мову одним словом, то, мабуть, «багатогранна» буде вдалим варіантом.



З іншого боку, «гнучка» — теж звучить непогано.

Хоч би там як, а вивчення мови Python — процес творчий і, як правило, тривалий. Ази, основи мови, можна опанувати досить легко. Але щодо «тонкощів», то це процес тривалий — правда, багато в чому цікавий і неочікуваний. Якоїсь миті може здатися, що всі крапки над «і» вже розставлено. Але не так усе просто! Обов'язково знайдеться цікавий приклад, зрозуміти який одразу може бути складно. Іноді доводиться переосмислити все, що до цього дізнався.

Ми під час вивчення мови Python намагалися зосередитися на головному, відокремити, образно висловлюючись, зерна від полови. При цьому деякі моменти залишилися «за бортом». Звісно, так не дуже добре. Проте ми себе виправдовуємо тим, що завдання стоять складні: зрозуміти й опанувати принципи досить нетривіальної й не дуже «стандартної» мови програмування. Тут, як кажуть, усі засоби гарні. Наскільки гарні наші засоби, покаже час. Але ось що треба чітко розуміти — так це те, що вчитися мові програмування (особливо такій, як Python) треба постійно, і краще на практичних прикладах. Щоб не бути голослівними, проілюструємо сказане — на прикладі, звісно.

Ми знаємо, що в аргументів функцій можуть бути значення за замовчуванням. Більше того, ми цією особливістю функцій не раз користувалися. Розглянемо функцію з таким програмним кодом:

```
def f(A=[0]):  
    A[0] += 1  
    return A[0]
```

Тут у функції `f()` оголошено один аргумент `A`, в якого є значення за замовчуванням — це список `[0]`, що складається всього з одного елемента (число 0). У загальному випадку передбачається, що аргумент `A` функції `f()` — список. У тілі функції командаю `A[0] += 1` на одиницю збільшується значення першого елемента в списку, переданому аргументом функції. Це нове значення елемента повертається як результат функції.

Що можна очікувати від такої функції? Можна очікувати, що список, переданий аргументом функції, буде змінено. Це справді так. Наприклад, якщо список `B=[10, 20, 30]`, то результатом виразу `f(B)` буде число 11, а список `B` після виконання команди дорівнюватиме `[11, 20, 30]` (точніше, змінна `B` буде посилятися на такий список). Також можна очікувати, що якщо ми викличемо функцію `f()` без аргументів, то результатом буде число 1. Це так. Але тільки перший раз. Якщо ми ще раз обчислимо вираз `f()`, у результаті отримаємо значення 2, потім — значення 3 і так далі. Хто має бажання, може зробити експеримент: після опису функції `f()` виконати оператор циклу:

```
for s in range(10):  
    print(f(), end=" ")
```

У результаті в області виводу з'явиться рядок із десяти натуральних чисел:

1 2 3 4 5 6 7 8 9 10

Отже, викликана без аргументів, функція `f()` кожен раз повертає нове значення — на одиницю більше за попереднє. Яка причина? А причина в значенні за замовчуванням для аргументу функції. Якщо конкретніше, то ось що відбувається. При створенні об'єкта для функції `f()` у пам'яті

виділяється місце під список [0]. Це неначе значення аргументу за замовчуванням. «Неначе» — тому що насправді значенням за замовчуванням є посилання на список. Це посилання залишається незмінним від запуску до запуску функції. Але сам список змінюється кожний раз, коли викликається функція. Звідси й такий трохи неочікуваний результат (хоча цілком логічний, якщо в усьому розібрatisя).

Наступна ілюстрація буде швидше про те, як при створенні програмних кодів мовою Python уносити в процес «творче начало». Отже, припустімо, є опис двох класів. Клас A:

```
class A:
    def show(self):
        print("Це клас A!")
```

У цьому класі всього один метод `show()`, яким відображується повідомлення. Щодо класу B, то він практично такий самий, як і клас A:

```
class B:
    def show(self):
        print("Це клас B!")
```

Ім'я класу — це насправді змінна, яка посилається на об'єкт класу. На об'єкт класу може посыматися більше, ніж одна змінна. Наприклад, законною є команда `MyClass=A`. У результаті змінна `MyClass` буде посыматися на той же об'єкт класу, що й змінна `A`. Іншими словами, змінна `MyClass` є «синонімом» назви класу `A`. Тепер шляхом наслідування створюємо клас `Stranger`:

```
class Stranger(MyClass):
    def show(self):
        MyClass.show(self)
        print("Клас Stranger!")
```

Тут дві особливості. По-перше, як ім'я базового класу ми зазначили змінну `MyClass`, а не `A`. По-друге, під час перевизначення методу `show()` у класі `Stranger` викликається версія цього методу з базового класу, причому в посиланні на цей метод явно вказано ім'я класу `MyClass`.

Далі за допомогою команди `obj=Stranger()` створюємо екземпляр `obj` класу `Stranger`. Якщо після цього виконати команду `obj.show()`, результатом будуть такі повідомлення:

Це клас А!
Клас Stranger!

А потім ми виконуємо команду `MyClass=B`, яка посилання в змінній `MyClass` із об'єкта класу А перекидає на об'єкт класу В. Після всіх означених маніпуляцій результатом команди `obj.show()` будуть повідомлення:

Це клас В!
Клас Stranger!

Хто бажає, може подумати, чому так відбувається. Можливо, у процесі пошуку відповіді на це запитання читач знайде для себе щось нове й цікаве. Адже пошук відповідей на запитання — найкращий шлях для саморозвитку.

Запитання і відповіді

Існує низка запитань загального характеру, які напрямки не стосуються мови Python (однак стосуються програмування як такого) і які досить часто виникають у тих, хто вчиться програмувати. Відповіді на деякі запитання наведені нижче.

Яка найпопулярніша мова програмування?

Однозначної відповіді на це запитання не існує. Адже популярність мови програмування можна визначати по-різному. Наприклад, можна визначати популярність мови за оголошеннями роботодавців, які підбирають для роботи програмістів, чи за кількістю програмістів, які використовують її в роботі, або за об'ємом написаного на мові програмного коду. Тому в залежності від методики оцінювання найбільш популярними можуть виявлятися різні мови. Однак існують мови, які при різних опитуваннях незмінно потрапляють в лідери рейтингу. Серед них Java, C++, C#, JavaScript і, безумовно, Python.

Чи потрібно попередньо знати якусь мову програмування, щоб вивчити Python?

Ні, такої потреби немає. Вивчення мови програмування умовно можна розбити на два блоки: це знання синтаксису мови й уміння складати алгоритми. Алгоритмічне мислення — уміння досить універсальне, яке жорстко не прив'язане до якоїсь конкретної мови програмування (хоча мова програмування накладає певні обмеження на характер і стиль програмних кодів). Синтаксис мови Python багато в чому унікальний, тому навіть для програмістів із практикою він виявиться незнайомим. Хоча, з іншого боку, наявність досвіду роботи з різними мовами програмування є «плюсом».

З якої мови краще почати вивчення програмування?

Слід врахувати два моменти. По-перше, важливою є мета, з якою вивчається мова. Скажімо, якщо хтось бажає вивчити мову аби отримати гарну роботу, то розумно почати з тієї мови, яка найбільш запотребована на ринку праці на даний момент. По-друге, в кожній мові програмування є те, що називається «порогом входження». Простіше кажучи, завжди існує певний психологічний та інтелектуальний бар'єр, який потрібно здолати при вивченні мови програмування. Наприклад, для мов Java і C# цей бар'єр вищий, ніж для мови C++. Для мови Python він досить низький, так що Python є непоганим «кандидатом» на роль першої мови програмування.

Чи існують вікові обмеження щодо навчання програмуванню?

Ні, вікових обмежень немає. Програмувати можуть усі, починаючи з підлітків (а іноді й дітей) і закінчуючи людьми похилого віку. Питання лише у бажанні й мотивації.

Чи можна навчитись програмувати самостійно?

Так, це можливо. На сьогодні існує значна кількість книг, методичних розробок, онлайн-курсів та інших інформаційних ресурсів, які дозволяють самостійно опанувати будь-яку мову програмування. Разом із тим слід розуміти, що спілкування з фахівцями, обмін досвідом — надзвичайно корисний процес, який дозволяє зростати професійно. Тому за можливості треба підтримувати контакти з колегами і намагатися «зануритися» у програмне середовище.

З чого краще починати навчання програмуванню?

Існують різні методики. Можна почати з відвідування лекцій (якщо є така можливість). Причому це можуть бути як звичайні лекції на курсах чи в навчальному закладі, так і вебінари, онлайн-лекції чи відеоуроки. Непоганий варіант — вибрати книгу з відповідної

мови програмування (бажано спеціальне видання для новачків) і почати працювати з матеріалом книги. Корисними будуть різноманітні інтернет-форуми, на яких програмісти-професіонали консультирують програмістів-початківців і відповідають на їхні запитання. Та головне — слід пам'ятати, що запорукою успіху у вивченні будь-якої мови програмування є постійна практична робота з програмним кодом. Не можна обмежуватись лише теоретичним вивченням матеріалу.

Як покращити свої навички в програмуванні?

Після досягнення певного рівня бажано взяти участь у реальному проекті. Зазвичай це ті проекти, з якими доводиться мати справу за основним місцем роботи (якщо робота пов'язана з програмуванням). Тому при виборі місця роботи для багатьох програмістів важливим є не тільки рівень заробітної плати, але й характер роботи, тип задач, які доведеться розв'язувати, можливість кар'єрного росту. Якщо ж основна робота не пов'язана з програмуванням, то є сенс узяти участь в якомусь некомерційному проекті (в тому числі й у власному). Слід також мати на увазі, що існують біржі фрілансу, на яких цілком реально знайти цікавий проект для реалізації.

Чи необхідно добре знати математику, аби навчитись програмувати?

У принципі, такої необхідності немає. Однак потрібно мати на увазі, що математична підготовка для програміста — це майже як фізична підготовка для футболіста. В історії футболу є по-справжньому великі футболісти, які перебували в не найкращій фізичній формі. Однак це скоріше виключення з правил. Більшість професійних футболістів приділяють серйозну увагу фізичній підготовці. Наявність гарного математичного бекграунду у програміста значно розширює його професійні можливості й сприяє професійному зростанню.

Скільки мов програмування має знати програміст?

Формально достатньо знати одну. Та, як правило, цим справа не обмежується. Причин декілька. Одна з головних пов'язана з тим, що чим більше мов знає програміст, тим він є більш конкурентним на ринку праці. Також не слід відкидати і той факт, що вивчення мов програмування — це насправді дуже цікаво. Особливо якщо врахувати, що важко вивчити першу мову. Після цього процес знайомства з іншими мовами програмування відбувається набагато простіше.

Чи потрібно мати профільну освіту, аби стати програмістом?

Значна кількість програмістів-професіоналів має непрофільну освіту. Тобто аби стати програмістом, профільну освіту отримувати не обов'язково (хоча якщо така освіта є, то стати до лав програмістів-професіоналів дещо легше). Тут слід урахувати три моменти. По-перше, роботодавців, як правило, цікавлять реальні знання та здобутки претендента на посаду програміста, а не те, що у нього написано в дипломі (і чи є в нього диплом узагалі). Тому в більшості випадків на співбесідах перевіряють реальні вміння, а не записи в дипломах. По-друге, на сьогодні існують великі можливості для отримання необхідних фахових навичок навіть поза межами профільних навчальних закладів. Це самоосвіта, курси з програмування, онлайн-курси, відеолекції. До того ж практично будь-яка навчальна програма природничого профілю в університеті містить значну кількість комп'ютерних навчальних курсів. По-третє, сучасні тенденції на ринку програмних технологій такі, що великі комп'ютерні компанії займаються не просто створенням програмних продуктів, але фактично надають ще й консалтингові послуги. Аби зробити успішну кар'єру в таких компаніях необхідно не просто вміти програмувати, бажано ще й мати широкий кругозір. Тому, наприклад, випускники фізичних факультетів при прийомі на роботу на посаду програмістів цінюються не менше, ніж претенденти, які закінчили профільний навчальний заклад.

Навчальне видання

ВАСИЛЬСВ Олексій Миколайович ПРОГРАМУВАННЯ МОВОЮ РУTHON

Головний редактор Богдан Будний

Редактор Володимир Дячун

Дизайн обкладинки Андрія Кравчука

Технічний редактор Неля Домарецька

Комп'ютерна верстка Андрія Кравчука

Підписано до друку 20.09.2018. Формат 70×100/16.

Папір офсетний. Гарнітура Century Schoolbook.

Умовн. друк. арк. 40,95. Умовн. фарбо-відб. 40,95.

Видавництво «Навчальна книга — Богдан»
Свідоцтво про внесення суб’єкта видавничої справи
до Державного реєстру видавців, виготовників
і розповсюджувачів видавничої продукції
ДК №4221 від 07.12.2011 р.

Видавництво «Навчальна книга — Богдан» у соцмережах:

 bohdanbooks  bohdan_books

 YouTube c/NKBohdan  t.me/bohdanbooks