PYTHON у прикладах і задачах. Частина 1. Структурне програмування Навчальний посібник

Book · October 2017

CITATIONS
0

READS
1,901

1 author:

Andrii Krenevych
National Taras Shevchenko University of Kyiv
24 PUBLICATIONS **25** CITATIONS

SEE PROFILE

Київський національний університет імені Тараса Шевченка

КРЕНЕВИЧ А.П.

PYTHON
У ПРИКЛАДАХ І ЗАДАЧАХ
Частина 1. Структурне програмування

Навчальний посібник

КИЇВ 2017

УДК 004.438С(075.8); 519.682

Рецензенти:
доктор фіз.-мат. наук А.С. Олійник (КНУ імені Тараса Шевченка)
кандидат техн. наук С.М. Алхімова (НТУУ «КПІ»)

Креневич А.П.

Python у прикладах і задачах. Частина 1. Структурне програмування Навчальний посібник із дисципліни "Інформатика та програмування" – К.: ВПЦ "Київський Університет", 2017. – 206 с.

Посібник призначений для практичного опанування програмування із застосуванням мови Python. Він охоплює основні розділи структурного програмування, що викладаються у вищих навчальних закладах для студентів математичних, природничих та інженерних спеціальностей.

У посібнику у систематизованому вигляді наводяться короткі теоретичні відомості, типові приклади розв'язання задач і задачі для самостійної роботи. Посібник складено з урахуванням досвіду викладання програмування на механіко-математичному факультеті Київського національного університету імені Тараса Шевченка.

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

ЗМІСТ

Вступ

Серед програмістів існує такий відомий рекурсивний жарт: «Для того, щоб навчитися програмувати потрібно програмувати». Тобто, практичному програмуванню неможливо навчитися теоретично – навчання полягає у розв'язанні великої кількості різноманітних задач із застосуванням комп'ютера.

Цей посібник є спробою викладення теоретичного та практичного матеріалу у обсязі достатньому для опанування студентами структурного програмування та отримання навичок, які у подальшому допоможуть студентам самостійно вивчати програмування.

Мовою програмування, що використовується у посібнику, є мова Python 3. Такий вибір мови програмування зумовлений її простотою, універсальністю, потужністю та широким розповсюдженням у розрізі сучасних інформаційних технологій. Проте, слід зауважити, що більшість завдань для самостійної роботи є інваріантними щодо мови програмування. Тому посібник може використовуватися як збірник задач з програмування на будь-якій мові.

Посібник складається з 10 основних параграфів. Кожен параграф присвячений конкретній темі структурного програмування, супроводжується теоретичним матеріалом у обсязі, достатньому для розв'язання задач, містить приклади розв'язання типових задач і перелік задач для самостійної роботи. Теоретичний матеріал використовує матеріали лекцій [2], що викладаються студентам механіко-математичного факультету протягом останніх років та рекомендуються до вивчення у якості додаткового матеріалу. Крім цього, під час написання посібника використовувався матеріал довідникової літератури, посібників та електронних видань з переліку [1, 3, 4, 5, 6].

Приклади розв'язання задач супроводжуються детальними описами алгоритмів, а ключові моменти програм пояснені у коментарях. Набір задач для самостійної роботи складений з урахуванням багаторічного досвіду викладання курсу програмування на механіко-математичному факультеті Київського національного університету імені Тараса Шевченка та частково базується на посібниках [7, 8]. Також у посібнику використовуються вправи, взяті з інших задачників та електронних джерел [2, 9, 10, 11, 12, 13, 14]. Слід зауважити, що задачі та приклади деяких розділів можуть використовувати матеріал з попередніх розділів. Це вимагає від читача послідовного опрацювання і засвоєння матеріалу.

Нумерація всіх об'єктів у посібнику (прикладів, задач, рисунків тощо) складається з двох частин: номеру параграфу та порядкового номеру об'єкта у цьому розділі. Багато з представлених у посібнику задач мають підпункти, що позначаються маленькими літерами англійського алфавіту. При посиланні на задачу зазначається її номер, перед яким

вказується слово "задача" (у потрібній формі). У посиланні також може бути позначений підпункт задачі.

Для запису абстрактних об'єктів (математичних та булевих змінних, векторів, матриць, рядків, абстрактних даних, математичних і алгоритмічних функцій тощо) у посібнику використовується математичний курсив. Для відображення фрагментів коду програм, інструкцій та ідентифікаторів, що використовуються при написанні програм, застосовується моноширинний шрифт з підсвічуванням (різними кольорами), аналогічним до того, яке використовується у сучасних інтегрованих середовищах розробки. Ці заходи мають значно полегшити процес сприйняття матеріалу.

Автор висловлює щиру подяку колегам кафедри математичної фізики доцентам Обвінцеву О.В., Довгому Б.П. та Вакалу Є.С. за корисні поради, конструктивну критику та допомогу при створенні цього посібника.

§0 БАЗОВІ ПОНЯТТЯ ПРОГРАМУВАННЯ

Цей курс присвячений програмуванню, тобто написанню алгоритмів, що будуть виконуватися комп'ютером. Програмування є однією зі складових інформатики. Тому спочатку розглянемо її основні поняття.

Базовим поняттям інформатики є поняття **інформація**. Це поняття відіграє в інформатиці настільки ж важливу роль, як поняття числа в алгебрі або поняття точки в геометрії. Термін інформація походить від латинського слова «informatio», яке має декілька значень: роз'яснення; виклад фактів, подій; витлумачення; представлення, поняття; ознайомлення, просвіта. Це базове аксіоматичне поняття у інформатиці – його не можна подати через інші «простіші» поняття.

Інформатика (eng. computer science) – це наука про методи зображення, накопичення, передачі і обробки інформації за допомогою комп'ютерів.

Інформація в інформатиці передбачає наявність

- матеріального носія інформації,
- джерела і передавача інформації,
- приймача інформації,
- каналу зв'язку між джерелом і приймачем інформації.

Виконавець – пристрій, здатний виконувати дії із заданого набору. Команду на виконання окремої дії звичайно називають **інструкцією** (або **оператором**). Найпоширенішими прикладами виконавців є комп'ютери. Іншими прикладами виконавців можуть бути побутові прилади, роботи, персональні гаджети.

Алгоритм – набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій.

Комп'ютерна програма – запис алгоритму у формі, придатній для виконання комп'ютером.

Програмування – процес побудови комп'ютерних програм.

Розробка програмного забезпечення – комплексний процес створення програмного забезпечення, що включає в себе проектування, програмування, тестування та підтримку програмного забезпечення.

Мова програмування – формальна знакова система призначена для опису алгоритмів та структур даних, засобами якої можна виражати алгоритми для виконання їх за допомогою комп'ютера. Вона визначає набір лексичних, синтаксичних і семантичних правил, що задають зовнішній вигляд програми і дії, які виконує комп'ютер під її управлінням.

Отже мова програмування це штучна мова, що використовується для передачі команд комп'ютерам.

Вихідний код (англ. source code, також використовуються терміни програмний код, текст програми) – набір інструкцій написаних мовою програмування у формі, що її може прочитати і модифікувати людина.

Компілятор – комп'ютерна програма що перетворює (компілює) вихідний код, написаний мовою програмування високого рівня, на семантично еквівалентний код на мові програмування низького рівня, який, необхідний для виконання програми комп'ютером.

Особливості:

- компілятор транслює (компілює) у машинний код (бінарний файл) всю програму без її виконання;
- компіляція здійснюється тільки один раз в середовищі розробника, після чого той же бінарний файл можна розповсюдити на машини користувача, де він може бути виконаний без додаткового перекладу;
- для запуску скомпільованої програми, як правило, не потрібно додаткових технічних засобів.

Інтерпретатор – програма, що здійснює покомандний аналіз програми, її обробку, перетворення у машинний код, та виконання.

Особливості:

- програма поширюється у вигляді вихідного коду;
- те, що вихідний код легко прочитати і скопіювати, може бути проблема з точки зору авторського права;
- програма має бути трансльована на кожній машині, що займає більше часу, але робить розповсюдження програми незалежним від архітектури машини;
- для виконання програми, необхідно, щоб на машині була встановлена програма-інтерпретатор.

Основні етапи побудови програми такі

- розробка алгоритму;
- введення тексту програми – створення вихідного коду;
- перевірка синтаксичної правильності;
- компіляція (не потрібна для інтерпретованих програм);
- виконання програми.

Імперативне програмування - це парадигма програмування (стиль написання вихідного коду комп'ютерної програми), для якого характерним є таке:

- у вихідному коді програми записуються інструкції (команди);
- інструкції виконуються послідовно;
- дані, отримані при виконанні інструкції, можуть записуватися в пам'ять;
- при виконанні інструкції дані, отримані при виконанні попередніх інструкцій, можуть зчитуватися з пам'яті.

Імперативна програма це послідовність наказів (eng. imperative-наказ), які повинен виконати комп'ютер. Поширений синонім для терміну «імперативне програмування» – процедурне програмування.

Мова програмування Python, що є предметом вивчення цього курсу, підтримує парадигму імперативного програмування.

Структурне програмування – це парадигма програмування, згідно з якою програми має вигляд ієрархічної структури блоків. Відповідно до структурного програмування будь-яка програма є структурою, побудованою з трьох типів базових конструкцій:

- послідовного виконання – тобто одноразового виконання інструкцій у тому порядку, в якому вони записані у тексті програми;
- розгалуження – одноразового виконання однієї з двох інструкцій, в залежності від виконання деякої заданої умови;
- циклу - багаторазового виконання однієї і тієї ж інструкції.

У програмі базові конструкції можуть бути вкладені одна в одну довільним чином. Жодних інших засобів керування послідовністю виконання операцій у структурному програмуванні не передбачається.

§1 ЛІНІЙНІ ПРОГРАМИ

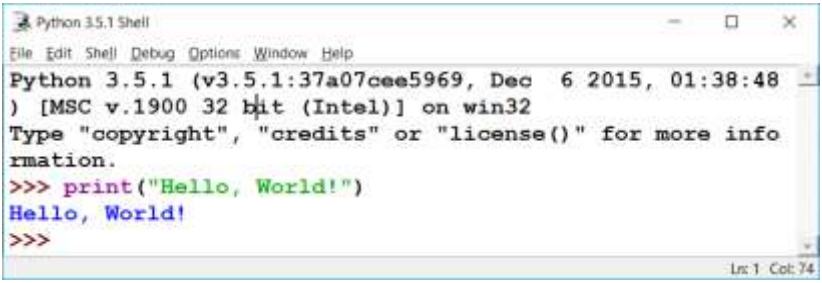
Перша програма на Python

Традиційно, першою програмою при вивченні будь-якої мови програмування є програма «Hello, World!». Ця програма виводить на екран (або в консоль) привітання «Hello, World!». Щоб написати її у Python, досить однієї інструкції

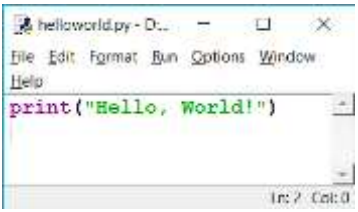
```
print("Hello, World!")
```

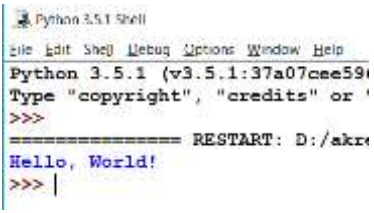
Виконати цю програму можна двома способами

- Ввести у консолі Python код програми (наприклад у інтерактивному режимі IDLE після символів очікування >>>) і натиснути клавішу Enter:



- Зберегти код програми у (текстовому) файлі з розширенням .py та запустити його на виконання. Наприклад, у IDLE для створення файлу необхідно виконати ланцюг File → New File з меню (або натиснути Ctrl + N). Для запуску програми необхідно виконати ланцюг Run → Run Module з меню (або натиснути клавішу F5)





Синтаксис

Синтаксис мови програмування це набір правил, що описує комбінації символів алфавіту, які вважаються правильно структурованою програмою або її фрагментом.

Отже, синтаксис визначає як буде виглядати програма на цій мові, зокрема, як пишуться оператори, оголошення і інші мовні конструкції.

Основні правила

Основні правила синтаксису мови програмування Python полягають у такому:

- Кінець рядка є кінцем інструкції;
- Допускається записувати кілька інструкцій у одному рядку. При цьому їх розділяють крапкою з комою

```
a = 1; b = 2; print(a, b)
```

- Допускається записувати одну інструкцію у кількох рядках. Для цього її необхідно обмежити круглими, квадратними або фігурними дужками:

```
y = (x**5 + x**4 +
        x**3 + x**2 + x + 1)
```

- Вкладені інструкції об'єднуються в блоки за величиною відступів. Для відступу, як правило використовують 4 пробілу;
- Вкладені інструкції в Python записуються відповідно до одного і того ж шаблону, коли основна інструкція завершується двокрапкою, слідом за якою розташовується вкладений блок коду, зазвичай з відступом під рядком основної інструкції

```
Основна інструкція:
    Вкладений блок інструкцій
```

Ідентифікатор – це послідовність символів, що може використовуватися як ім’я елементу (об’єкту) у мові програмування.

У Python у ролі ідентифікаторів може використовуватися послідовність символів, що складається з літер латинського алфавіту (при цьому розрізняються великі та маленькі літери), цифр та знаку

нижнього підкреслення «_», при умові, що першим символом не є цифра.

Приклади ідентифікаторів у Python:

```
y
_error777
variable__111
```

Константа – спосіб адресації даних, зміна яких програмою не передбачається або забороняється.

Константи поділяються на два види – **літерали** і **іменовані константи. Літерал** – стале значення певного типу даних, записане у вихідному коді комп'ютерної програми.

```
256              # Числовий літерал цілого типу
0.1              # Числовий літерал дійсного типу
3.14159          # Числовий літерал для числа пі
"Рядок-літерал"  # Рядковий літерал
```

Як випливає з означення літерали розрізняються за типами. Типи літералів визначають не лише зображення літералу, але й операції, які можна проводити над цими літералами. Наприклад, для літералів числових типів визначені арифметичні операції, операції порівняння тощо.

Рядкові літерали беруть у апострофи, подвійні лапки, потрійні апострофи або потрійні подвійні лапки, для того, щоб відрізняти їх від ідентифікаторів.

```
'Рядок-літерал'

"Рядок-літерал"

'''Це рядок-літерал, який можна
   розміщувати у кількох рядках'''

"""Це також рядок-літерал, який можна
   розміщувати у кількох рядках"""
```

Іменована константа відрізняється від літералу тим, що крім сталого значення вона ще має ім'я. Таким чином, замість значення літералу можна використовувати це ім'я. Це зручно, наприклад, в тому разі, коли певний літерал використовується багаторазово або має велику довжину. Слід зауважити, що у Python немає окремої конструкції

для створення іменованих констант. Тому іменовані константи позначаються аналогічно до змінних.

```
pi = 3.14159      # pi - іменована константа для числа пі
```

Змінна – іменована область пам'яті, ім'я якої (ідентифікатор) використовується для здійснення доступу до даних (значення змінної), що містяться у цій області пам'яті.

Значення змінної є літералом. Тип змінної визначається типом літералу, що є значенням цієї змінної.

```
x = 23      # x - змінна зі значенням 23
y = x + 3   # y - змінна зі значенням 26
```

При роботі зі змінними у Python потрібно пам'ятати, що змінна є посиланням на область пам'яті, де зберігаються її дані. Наприклад, якщо

```
a = 3
s = a
```

то це фактично означає, що використовується не дві різних змінних `a` та `s`, а одна змінна зі значенням 3, що має два різних імені `a` та `s`.

Змінні у Python поділяються на два види:

- **змінювані** (**mutable**)
- **незмінювані** (**immutable**).

До об'єктів змінюваних типів є безпосередній доступ для модифікації даних цих об'єктів (без створення нових). Модифікація даних об'єктів незмінюваних типів можлива лише через створення нових об'єктів (тобто через виділення пам'яті).

Змінні всіх простих типів (`int`, `float`, `complex`) належать до незмінюваних. При вивченні нових типів даних будемо зауважувати, до якого виду відноситься кожен з них.

Ключові слова мови програмування – це зарезервовані ідентифікатори, що наділені певним сенсом. Їх можна використовувати виключно відповідно до значення яке закріплене за ними у мові програмування.

Зауваження. Оголошуючи новий ідентифікатор у якості імені змінної, функції або іншого об'єкту, потрібно переконатися, що цей ідентифікатор не є ключовим словом мови програмування. Інакше програма або не виконається, або виконається з помилками.

Сучасні інтегровані середовища програмування, як правило ключові слова виділяють іншим кольором. Тому пишучи програму, ви скоріше за все зрозумієте, що цей ідентифікатор не можна використовувати у ваших цілях. Наприклад, у коді нижче ідентифікатори **while** і **if** є ключовими словами.

```
while i <= N - 1:
    if N % i == 0:
        prime = False
    i = i + 1
```

Коментування коду

Коментарі – зрозуміла для програміста анотація, що знаходиться безпосередньо у вихідному коді комп'ютерної програми.

Коментарі пишуть для того щоб зробити код більш зрозумілим. Вони ігноруються при компіляції й інтерпретації. Коментарі також обробляють різними способами для створення окремої документації на базі вихідного коду за допомогою генераторів документації.

Коментарі поділяють на **однорядкові** та **багаторядкові**.

Однорядковий коментар може розташовуватися лише в одному фізичному рядку програми. Починається символом # і закінчується кінцем рядка.

```
x = 23      # Цей текст коментар
```

Багаторядковий коментар може розташовуватися у кількох фізичних рядках програми. Виділяється з двох боків потрійними апострофами або потрійними подвійними лапками

```
'''Багаторядковий коментар може
розташовуватися у кількох
фізичних рядках програми'''
```

Основні найпростіші команди

Тотожна інструкція

Тотожна інструкція задається командою

```
pass
```

Інтерпретатор не виконує жодних дій, коли зустрічає цю інструкцію та переходить до виконання наступної інструкції. Основне її призначення – використовуватися там, де інструкція вимагається синтаксисом мови, проте ніяких дій виконувати не потрібно.

Присвоєння

Присвоєння — одна з центральних конструкцій в імперативних мовах програмування.

Присвоєння – механізм у програмуванні, що дозволяє динамічно змінювати зв'язки об'єктів даних (наприклад, змінних) із їхніми значеннями.

Синтаксис присвоєння у Python такий

```
x = e
```

тут `e` – змінна (об'єкт) або арифметичний вираз (інструкція, що повертає результат), який може бути записаний у змінну. `x` – змінна, у яку записується результат `e`.

Після інструкції присвоєння попереднє значення змінної `x` стає недосяжним.

```
x = 12
y = (x**5 + x**4 + x**3 + x**2 + x + 1)
```

Тут змінній `x` присвоюється значення `12`, а змінній `y` значення арифметичного виразу, у якому бере участь змінна `x`.

Введення та виведення

Інструкції введення та виведення використовуються для взаємодії користувача та програми, а також для обміну інформацією між зовнішніми носіями та оперативною пам’яттю комп’ютера (об’єктами програми). Існують різноманітні способи введення і виведення

інформації з використанням клавіатури, екрану, файлів, звукової карти комп'ютера, принтера та інших периферійних пристроїв. Проте, наразі під введенням будемо розуміти введення користувачем певної інформації з клавіатури під час виконання програми, а під виведенням – друкуванням даних у консоль під час виконання програми.

Інструкція введення має такий синтаксис

```
x = input(S)
```

де `x` – змінна, у яку записується результат введення з клавіатури, `S` – рядок підказки.

Коли інтерпретатор зустрічає інструкцію введення, він призупиняє роботу програми і очікує, коли з клавіатури буде щось введено. При цьому на екран виводиться рядок підказки `S`. Результат введення записується у змінну, що стоїть зліва від знаку рівності.

Синтаксис інструкції виведення такий

```
print(S1,…,SN)
```

де `S1,…,SN` – вирази, що виводяться на екран. Виведення виразів відбувається через символ пропуску.

Для прикладу розглянемо таку просту програму

```
name = input("Як тебе звуть? ")
print("Привіт,", name)
```

Результат виконання цієї програми зображено на скріншоті. Як бачимо програма вивела повідомлення "Як тебе звуть? ", та дочекавшись введення з клавіатури, вивела повідомлення "Привіт, Андрій".

Інструкція виведення `print` має два необов'язкових параметри `sep` та `end`. Вони вказують які символи використовуються у якості розділювача та кінця рядка відповідно. Типовими значеннями є відповідно символ пропуску та кінець рядка. Для прикладу проаналізуйте результат виконання коду.

```
>>> print(1,2,3)
1 2 3
>>> print(1,2,3, sep="_")
1_2_3
```

```
>>> print(1); print(2)
1
2
>>> print(1, end="!"); print(2, end="!")
1!2!
```

Досить часто виникає ситуація, коли необхідно вивести рядок (повідомлення), згідно з заданим шаблоном, підставивши у визначені позиції дані, отримані у результаті виконання програми. Таку підстановку можна здійснювати за допомогою оператора `%`.

Оператор `%` підставляє значення виразів, що стоять справа від нього у визначені спеціальними символами позиції. Спеціальні символи це послідовності з двох символів, перший з яких символ "`%`". Таблиця спеціальних символів наведена нижче

Таблиця 1.1. Найуживаніші спеціальні символи

Формат	Тип

`%d`
`%i`
`%u`
	Десяткове число ціле число.

`%e` `%E`	Дійсне число в експоненційній формі.
`%f` `%F`	Дійсне число у звичайному форматі.
`%c`	Символ (або код символа).
`%s`	Рядок символів.
`%%`	Символ %.

Розглянемо на прикладах

```
>>> "First = %d, second = %d" % (1, 2)
First = 1, second = 2
```

Справа від оператора `%` стоїть у дужках послідовність з двох чисел 1 та 2. Як бачимо, інтерпретатор, замість першого спеціального символа "`%d`" вставив цифру 1, а замість другого – 2. Кількість підстановок за допомогою оператора `%` може бути довільна. Єдине правило, якого потрібно дотримуватися – кількість спеціальних символів у рядку зліва від оператора `%` має дорівнювати кількості значень справа від оператора `%`.

```
>>> print("%s було %d років, коли він пішов у %d клас" %
("Миколі", 7, 1))
```

```
Миколі було 7 років, коли він пішов у 1 клас
```

Якщо у операторі підстановці використовується лише одне значення дужки справа від оператора `%` можна опустити

```
>>> print("Високосний рік має %d днів" % 366)
Високосний рік має 366 днів
```

Оператор `%` можна використовувати скрізь де необхідно зробити підстановку, зокрема у його можна використовувати у інструкції введення `input`, наприклад,

```
x = input("Введіть %d-й член послідовності " % 10)
```

Лінійна програма

Лінійна програма – це програма, що складається лише з інструкцій введення, виведення, присвоєння та тотожної команди.

Числові типи даних

Числа у Python нічим не відрізняються від звичайних чисел та підтримують математичні операції відомі вам з математики. Числові типи у Python представлені трьома типами: цілими, дійсними та комплексними. Розглянемо детальніше кожен з них.

Цілі числа

Змінні та літерали цього типу набувають значень з (обмеженої) множини цілих чисел. Цілий тип у Python, при необхідності, позначають `int`.

Таблиця 1.2. Арифметичні операції для цілого типу

Операція	Опис
`x + y`	сума x та y
`x - y`	різниця x та y
`x * y`	добуток x та y
`x / y`	частка від ділення x на y
`x // y`	ділення націло x на y
`x % y`	остача від ділення x на y
`-x`	унарний мінус – повертає x протилежного знаку
`x ** y`	x піднесене до степеня y

```
>>> x = 1 + 2
>>> print(x);
3
```

```
>>> x = 11 // 2
>>> print(x);
5
```

```
>>> x = 2 ** 4
>>> print(x);
16
```

Таблиця 1.3. Базові функції для цілого типу

Функція	Опис
`abs(x)`	повертає модуль x
`int(x)`	результатом буде перетворення x до цілого
`float(x)`	результатом буде перетворення x до дійсного
`divmod(x, y)`	пара (x // y, x % y)

```
>>> x = int(9.6)
>>> x
9
```

```
>>> x = abs(-52)
>>> print (x)
52
```

Таблиця 1.4. Пріоритет операцій від найвищого до найнижчого

`**`
`*, /, //, %`
`+, -`

Пріоритет унарного мінуса залежить від того, з якою операцією він використовується.

Для прикладу розглянемо такий код

```
>>> -3 ** 2
-9
>>> 3 ** -2
0.1111111111111111
```

Як бачимо у першому випадку початку виконується операція піднесення до степеня, а вже потім застосовується унарний мінус, а в другому випадку навпаки.

Пріоритет арифметичних операцій, можна змінювати за допомогою дужок:

```
>>> 2 + 2 * 2
6
>>> (2 + 2) * 2
8
```

Приклад 1.1. Знайти суму цифр двозначного числа.

Розв'язок. Нехай x – задане двозначне число, наприклад 25. Сума цифр цього числа є сумою першої та другої цифр цього числа: $2 + 5 = 7$. Першу цифру цього числа можна знайти як ділення числа x націло на 10. Друга ж цифра цього числа є остачею від ділення числа x на 10. Тоді програма буде мати вигляд

```
x = int(input("Введіть двозначне число "))
first = x // 10  # first - перша цифра числа
second = x % 10  # second - друга цифра числа
suma = first + second
print("Сума цифр числа %d = %d" % (x, suma))
```

Зауважимо, що операція input повертає літерал рядкового типу. Тому у першому рядку програми, відбувається перетворення результату введення з клавіатури до цілого числа за допомогою інструкції int.

Результатом виконання цієї програми для введеного числа 25 буде

```
Введіть двозначне число 25
Сума цифр числа 25 = 7
```

Дійсні числа

Дійсні числа у Python записуються у вигляді десяткового дробу, для розділення цілої і дробової частини у якому застосовується символ крапка. Дійсний тип у Python, позначають `float`

```
>>> 3.11
3.11
```

Щоб відрізняти літерали дійсних чисел без дробової частини від цілих, записують десятковий дріб з нульовою дійсною частиною

```
>>> 3.0
3.0
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
```

Функція `type` повертає тип літералу. Як бачимо в першому випадку цілий тип, а в другому – дійсний.

Крім зображення дійсних чисел у вигляді звичайного десяткового дробу використовується експоненціальних або показниковий запис.

Експоненціальний запис (формат) – зображення дійсних (дробових) чисел у вигляді мантиси і порядку:

$$N = M \cdot n^p,$$

де N – число, M– мантиса, n – основа показникової функції, p – порядок.

В інформатиці основу показникової функції n беруть рівною 10, а комп'ютерну реалізацію чисел у експоненціальній формі називають *числами з плаваючою комою* (*плаваючою крапкою*). Показник степеня в обчислювальних машинах прийнято відділяти від мантиси символами `"E"` або `"e"` (скорочення від "exponent"). Наприклад, число

$$1.234 \cdot 10^{-56}$$

записується так:

```
1.1234e-56
```

Слід зауважити, що зображення дійсного числа в показниковій формі є неоднозначним. Наприклад, число 0.0001 у показниковій формі можна записати багатьма способами, наприклад,

$$0.0001 = 10 \cdot 10^{-5} = 0.1 \cdot 10^{-3}.$$

Тому в інформатиці використовують нормалізовану форму запису чисел з плаваючою комою.

Нормалізованою формою запису числа плаваючою комою називають зображення, у якому мантиса належить проміжку $[1,10)$.

У такій формі будь-яке число (крім нуля) записується єдиним чином. Наприклад, вищенаведене число 0.0001 у нормалізованій формі буде мати вигляд:

$$0.0001 = 1 \cdot 10^{-4}.$$

Над змінними та літералами дійсного типу можна проводити ті ж арифметичні операції, що і для цілого типу наведені у таблицях 1.2 та 1.3. Пріоритет арифметичних операцій також зберігається згідно з таблицею 1.4.

```
>>> 3.0 * 2.0
6.0
```

```
>>> 9.0 ** 0.5
3.0
```

Слід зауважити, що через двійкове кодування дійсних чисел у пам'яті комп'ютера, дійсні числа не є точними, що може призводити до неочікуваних, на перший погляд користувача, результатів.

Розглянемо на прикладі.

```
>>> res1 = 0.5+0.5
>>> res2 = 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
```

З точки зору математики змінні `res1` та `res2`, мають містити однаковий результат. Проте, вивівши їхні значення на екран

```
>>> res1
1.0
>>> res2
0.9999999999999999
```

переконуємося, що це не так.

Оперуючи дійсними числами завжди потрібно враховувати похибку арифметичних операцій.

При роботі з числовими типами даних, Python, без додаткових застережень (якщо це допустимо математичними правилами), допускає використання у арифметичних операціях операндів різних числових типів, наприклад, дійсного і цілого. У такому арифметичному виразі всі

операнди (або результат при необхідності) будуть автоматично зведені до типу з найширшими можливостями серед представлених.

```
>>> 4.5 + 4
8.5
>>> 9 ** -1
0.1111111111111111
```

Приклад 1.2. Обчислити значення многочлена

$$y = x^6 - 4x^4 + 3x - 7$$

для заданого значення x.

Розв'язок. Програма буде мати вигляд

```
x = float(input('x=? '))
y = x ** 6 - 4 * x ** 4 + 3 * x - 7
print('y=', y)
```

Окрім основних арифметичних операцій та функцій наведених вище, Python надає програмісту математичну бібліотеку `math`, яка містить додаткові математичні функції і сталі. Для використання бібліотеки її необхідно імпортувати командою

```
import math
```

Команди імпортування бібліотек, як правило, записують на початку файлу, що містить вихідний код. Проте можна проводити імпорт бібліотеки безпосередньо перед використанням функцій з неї.

Таблиця 1.5. Функції та сталі з бібліотеки math

Функція	Опис
`math.pi`	Константа $\pi = 3.141592...$
`math.e`	Константа $e = 2.718281...$
`math.sqrt(x)`	Обчислення функції $\sqrt{x}$
`math.exp(x)`	Обчислення функції e^x
`math.log(x)`	Обчислення функції $\ln x$
`math.log(x, a)`	Обчислення функції $\log_a x$
`math.log1p(x)`	Обчислення функції $\ln(1+x)$
`math.log2(x)`	Обчислення функції $\log_2 x$
`math.log10(x)`	Обчислення функції $\lg x$
`math.cos(x)`	Обчислення функції $\cos x$
`math.sin(x)`	Обчислення функції $\sin x$

Функція	Опис
`math.tan(x)`	Обчислення функції $\mathrm{tg}\, x$
`math.acos(x)`	Обчислення функції $\arccos x$
`math.asin(x)`	Обчислення функції $\arcsin x$
`math.atan(x)`	Обчислення функції $\mathrm{arctg}\, x$
`math.atan2(y, x)`	Обчислення функції $\mathrm{arctg}\frac{y}{x}$
`math.cosh(x)`	Обчислення функції $\mathrm{ch}\, x$
`math.sinh(x)`	Обчислення функції $\mathrm{sh}\, x$
`math.tanh(x)`	Обчислення функції $\mathrm{th}\, x$
`math.hypot(x, y)`	Обчислення $\sqrt{x^2+y^2}$
`math.degrees(x)`	Повертає результат перетворення x з радіан у градуси
`math.radians(x)`	Повертає результат перетворення x з градусів у радіани
`math.factorial(n)`	Обчислення n! для цілого та невід'ємного значення аргументу.
`math.trunc(x)`	Дійсне число, яке є результатом відкидання дробової частини x
`math.ceil(x)`	Найменше ціле, яке більше або дорівнює x
`math.floor(x)`	Найбільше ціле, яке менше або дорівнює x

Приклад 1.3. Знайти кути заданого прямокутного трикутника.

Розв'язок. Будемо вважати, що прямокутний трикутник задається двома катетами a та b. Тоді, один з його гострих кутів визначається як

$$\beta = arctan\frac{a}{b}.$$

Отже, програма буде мати вигляд

```
import math              # імпорт математичної бібліотеки

a = float(input("Введіть перший катет "))
b = float(input("Введіть другий катет "))
beta = math.atan2(a, b)      # один з кутів у радіанах
betaDegree = math.degrees(beta) # перетворення у градуси

print("Перший кут = ", betaDegree)
print("Другий кут = ", 90.0 - betaDegree)
```

Результатом виконання цієї програми буде (наприклад, для двох однакових катетів одиничної довжини):

```
Введіть перший катет 1
```

```
Введіть другий катет 1
Перший кут =  45.0
Другий кут =  45.0
```

Комплексні числа

Комплексні числа у Python задають парою – дійсною і уявною частиною. Для позначення уявної частини, поруч з її значенням використовують символ `j`. Комплексний тип у Python, позначають `complex`

```
>>> c = 2 + 3j
>>> print(c)
(2+3j)
```

```
>>> c = complex(2,3)
>>> print(c)
(2+3j)
```

Вводити з клавіатури комплексні числа треба також у алгебраїчній формі використовуючи у якості уявної одиниці символ `j`

```
>>> c = complex(input("z = "))
z = 2+3j
>>> print(c)
(2+3j)
```

Для комплексних чисел у Python визначені чотири основні арифметичні операції, а також операція унарний мінус і піднесення до степеня.

Таблиця 1.6. Арифметичні операції для комплексного типу

Операція	Опис
`x + y`	сума x та y
`x - y`	різниця x та y
`x * y`	добуток x та y
`x / y`	частка від ділення x на y
`-x`	унарний мінус – повертає x протилежного знаку
`x ** y`	x піднесене до степеня y

Крім цього для комплексних чисел визначені такі базові функції

Таблиця 1.7. Базові функції для комплексного типу

Функція	Опис
`abs(z)`	модуль комплексного числа z
`complex(re, im)`	створення комплексного числа з пари дійсних чисел re та im.
`z.real`	дійсна частина числа z
`z.imag`	уявна частина числа z
`z.conjugate()`	комплексно-спряжене число до z

Розширений набір функцій для роботи з комплексними числами містить бібліотека `cmath`. Вона містить додаткові функції і сталі наведені у таблиці.

Таблиця 1.8. Функції та сталі з бібліотеки cmath

Функція	Опис
`cmath.pi`	Константа $\pi = 3.141592...$
`cmath.e`	Константа $e = 2.718281...$
`cmath.sqrt(z)`	Обчислення функції $\sqrt{x}$
`cmath.exp(z)`	Обчислення функції e^z
`cmath.log(z)`	Обчислення функції $\ln z$
`cmath.log(z, a)`	Обчислення функції $\log_a z$
`cmath.log10(z)`	Обчислення функції $\lg z$
`cmath.cos(z)`	Обчислення функції $\cos z$
`cmath.sin(z)`	Обчислення функції $\sin z$
`cmath.tan(z)`	Обчислення функції $\mathrm{tg}\, z$
`cmath.acos(z)`	Обчислення функції $\arccos z$
`cmath.asin(z)`	Обчислення функції $\arcsin z$
`cmath.atan(z)`	Обчислення функції $\mathrm{arctg}\, z$
`cmath.cosh(z)`	Обчислення функції $\mathrm{ch}\, z$
`cmath.sinh(z)`	Обчислення функції $\mathrm{sh}\, z$
`cmath.tanh(z)`	Обчислення функції $\mathrm{th}\, z$
`cmath.phase(z)`	Аргумент комплексного числа `z`
`cmath.polar(z)`	Зображення комплексного числа z у полярних координатах.
`cmath.rect(r,phi)`	Перетворення комплексного числа з полярними координатами `r`, `phi` у алгебраїчну форму.

Приклад 1.4. Доведемо зв'язок між трьома магічними сталими e, π, i:

$$e^{i\pi} = -1$$

Розв'язок. Скористаємося бібліотекою `cmath`

```
import cmath              # підключаємо модуль cmath

i = 1j                    # задамо уявну одиницю
pi = cmath.pi             # задамо число пі
z = cmath.exp(i * pi)
print(z)
```

Результатом виконання вищенаведеної програми буде виведення на екран числа

```
(-1+1.2246467991473532e-16j)
```

У якому уявна частина майже дорівнює нулю.

Задачі для аудиторної роботи

1.1. Вивести на екран таблицю

```
 x | 1 | 2 | 3 | 4 | 5
---+---+---+---+---+---
 y | 3 | 1 | 5 | 4 | 2
```

1.2. Дано натуральне тризначне число. Знайти суму цифр цього числа.

1.3. Обчислити гіпотенузу с прямокутного трикутника за катетами a та b.

1.4. Наближено визначити значення числа π, використовуючи ланцюговий дріб

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292}}}}.$$

1.5. Обчислити значення многочлена для введеного з клавіатури комплексного числа z:

a) $f(z) = 4z^4 + 3z^3 + 2z^2 + z + 1$; b) $f(z) = z^4 + 2z^2 + 1$;

Задачі для самостійної роботи

1.6. Вивести на екран рисунки:

a)
```
    *
  * * *
* * * * *
  * * *
    *
```

b)
```
* * * * *
* *   * *
*   *   *
* *   * *
* * * * *
```

c)
```
*******
*     *
* Hello *
*     *
*******
```

1.7. Вивести на екран текст:

a)
```
a   a   a
  a   a
a   a   a
```

b)
```
a-------a
|   a   |
a-------a
```

де `a` – введена з клавіатури цифра.

1.8. Зобразити на екрані декартову систему координат у вигляді

```
      ^ y

      |       x
------+-+------>
      | 1
      |
```

Зауваження. У цьому розділі під час написання програм вважати, що дані з клавіатури користувачем вводяться коректно.

1.9. Обчислити площу трикутника S за трьома сторонами a, b, c.

1.10. Знайти довжини всіх медіан, бісектрис і висот трикутника, якщо відомі три сторони a, b, c.

1.11. Обчислити відстань від точки (x_0, y_0) до:

a) заданої точки (x, y);

b) заданої прямої $ax + by + c = 0$;

c) точки перетину прямих $x + by + c = 0$ і $ax + y + c = 0$, де $ab \neq 1$.

1.12. Знайти об'єм циліндра, якщо відомо його радіус основи та висоту.

1.13. Знайти об'єм конуса, якщо відомо його радіус основи та висоту.

1.14. Знайти об'єм тора з внутрішнім радіусом r і зовнішнім радіусом R.

1.15. Вважаючи, що Земля має форму сфери радіуса $R = 6350$ км, знайти відстань до лінії горизонту від точки із заданою висотою h над Землею.

1.16. Без попередніх алгебраїчних перетворень обчислити значення арифметичних виразів:

a)

$$y = \frac{c + \dfrac{a}{a^2 + b^2}}{a + \dfrac{b}{b^2 + c^2}};$$

b)

$$y = 1 + \frac{1}{2 + \dfrac{1}{3 + \dfrac{1}{4}}}$$

1.17. Наближено визначити період обертання Землі навколо Сонця, використовуючи ланцюговий дріб

$$T = 365 + \frac{1}{4 + \dfrac{1}{7 + \dfrac{1}{1 + \dfrac{1}{3}}}}.$$

1.18. Скласти програми для розв'язання рівнянь:

a) $4.2343x + b = c$; b) $(5.23x + b)(d - 2.4y) = 0$;

c) $ax + b = cx + d$, $a - c \neq 0$,

де коефіцієнти a, b, c, d – вводяться з клавіатури.

1.19. Розв'язати квадратне рівняння $ax^2 + bx + c = 0$, де коефіцієнти a, b і c такі, що $b^2 - 4ac > 0$.

1.20. Дано дійсне число x. Користуючись тільки операцією множення, отримати:

a) x^4 за дві операції; b) x^6 за три операції;

c) x^9 за чотири операції; d) x^{15} за п'ять операцій;

e) x^{28} за шість операцій; f) x^{64} за шість операцій.

1.21. Не використовуючи операцію піднесення до степеня, обчислити значення многочлена для введеного з клавіатури значення x за найменшу кількість арифметичних операцій:

a) $y = x^4 + x^3 + x^2 + x + 1$; b) $y = x^4 + 2x^2 + 1$;

c) $y = x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$; d) $y = x^9 + x^3 + 1$;

e) $y = 16x^4 + 8x^3 + 4x^2 + 2x + 1$; f) $y = x^5 + x^3 + x$.

1.22. Скласти програму для обчислення значення многочлена від двох змінних для введеної з клавіатури пари чисел (x, y):

a) $f(x, y) = x^3 + 3x^2y + 3xy^2 + y^3$;

b) $f(x, y) = x^2y^2 + x^3y^3 + x^4y^4$;

c) $f(x, y) = x + y + x^2 + y^2 + x^3 + y^3 + x^4 + y^4$.

1.23. Скласти програму взаємного обміну значень цілих змінних x та y:

a) з використанням додаткової змінної;

b) без використання додаткової змінної.

1.24. Дано натуральне тризначне число. Знайти:

a) число одиниць, десятків і сотень цього числа;

b) число, утворене при прочитанні заданого числа справа наліво.

1.25. Дано натуральне тризначне число, у якому всі цифри різні. Знайти всі числа, утворені при перестановці цифр заданого числа.

1.26. Від тризначного числа x відняли його останню цифру. Після ділення результату на 10 до частки ліворуч дописали останню цифру числа x та отримали число n. За заданим числом n знайти вихідне число x. Вважати, що $10 < n < 999$, а число десятків у n не дорівнює нулю.

1.27. У тризначному числі x закреслено першу цифру. Якщо отримане число помножити на 10 і добуток додати до першої цифри числа x, то буде отримано число n. За заданим числом n, $1 < n < 999$ знайти число x.

1.28. Тіло починає рухатися без початкової швидкості з прискоренням a. Обчислити:

a) відстань, яку воно пройде за час t від початку руху;

b) час, за який тіло досягне швидкості v.

1.29. Обчислити кінетичну енергію тіла масою m, що рухається зі швидкістю v відносно поверхні Землі.

§2 РОЗГАЛУЖЕНІ ПРОГРАМИ

Розгалуження одна з базових конструкцій алгоритмів. За допомогою розгалужень, програма вибирає яку дію слід виконати, в залежності від значень змінних у момент перевірки.

Основи алгебри висловлювань

Основою розгалужень є алгебра висловлювань. Розглянемо її базові поняття.

Означення

Висловлювання (також використовують терміни логічний або булевий вираз**)** це твердження, якому завжди можна поставити у відповідність тільки одне з двох логічних (булевих) значень: Хибність (**`False`**) або Істина (**`True`**), кожне з яких є запереченням іншого

Приклади істинних висловлювань: «крокодил зелений», «$0 < 1$». Приклади хибних висловлювань: «усі чоловіки обожнюють автомобілі», «$2 + 2 = 5$».

Проте, у програмуванні використовують висловлювання, говорити про істинність або хибність яких можна лише у момент перевірки. Наприклад, висловлювання «$x < 1$» може бути як істинним так і хибним залежно від значення змінної x. Крім того, такі висловлювання можуть бути невизначеними. Такі висловлювання називаються умовами і будуть розглянуті нижче.

Нехай p і q – висловлювання. Над висловлюваннями визначено три базових логічних (булевих) операції:

Диз'юнкція (логічне «або» позначається **`or`**) – це бінарна операція, що діє за правилом: висловлювання p **`or`** q хибне тоді і тільки тоді, коли хибні одночасно висловлювання p і q, та істинне в інших випадках.

Кон'юнкція (логічне «і» позначається **`and`**) – це бінарна операція, що діє за правилом: висловлювання p **`and`** q істинне тоді і тільки тоді, коли істинні одночасно висловлювання p і q, та хибне в інших випадках.

Заперечення (логічне «не» позначається `not`) – це унарна операція, що діє за правилом: висловлювання `not` p є протилежним за змістом до висловлювання p.

Множина B = {`True`, `False`} над елементами якої визначені логічні операції диз'юнкція, кон'юнкція і заперечення називається **булевою алгеброю** або **алгеброю висловлювань**.

Умови

Відношення – це логічна операція, що ставить у відповідність кільком (як правило, двом) величинам булеве значення.

Прикладами відношень можуть бути висловлювання: «3 = 4», «0 < 1», «Микола вищий за Івана».

Операції відношення

У Python існує багато різних операцій відношення, наприклад, перевіри рівності об'єктів, приналежність об'єктів до певної колекції (множини), входження однієї множини до іншої тощо. Очевидно, що можливість застосування кожної операції залежить від типу об'єктів над якими вона проводиться. Наприклад, можна порівнювати дійсні числа на предмет яке з них більше, проте така операція для комплексних чисел не є коректною.

Вивчення операцій відношення є природнім у контексті вивчення типу даних, до елементів якого можуть застосовуватися ці операції. Тому при вивченні кожного нового типу будемо розглядати операції відношення, які можуть застосовуватися до об'єктів цих типів.

Розглянемо операції відношення для числових типів даних. Нехай вирази a і b належать до одного з числових типів – цілого, дійсного або комплексного. Тоді

- `==` (дорівнює) – бінарна операція, що діє за правилом: висловлювання `a == b` істинне тоді і тільки тоді, коли значення виразів `a` і `b` є однаковими;
- `!=` (не дорівнює) – бінарна операція, що діє за правилом: висловлювання `a != b` істинне тоді і тільки тоді, коли значення виразів `a` і `b` є різними;

Наприклад, відношення нижче буде завжди набувати значення **False**

```
5 == 6
```

Вищенаведені відношення можуть застосовуватися не лише для впорядкованих типів, але і для об'єктів будь-яких типів.

Нехай тепер вирази a і b належать до цілого або дійсного типу (тобто впорядкованого). Тоді

- < (менше) – бінарна операція, що діє за правилом: висловлювання `a < b` істинне тоді і тільки тоді, коли значення виразу є меншим за значення виразу;
- <= (менше або дорівнює) бінарна операція, що діє за правилом: висловлювання `a <= b` істинне тоді і тільки тоді, коли значення виразу a є меншим або дорівнює виразу `b`;
- > (більше) – бінарна операція, що діє за правилом: висловлювання `a > b` істинне тоді і тільки тоді, коли значення виразу a є більшим за значення виразу b;
- >= (більше або дорівнює) бінарна операція, що діє за правилом: висловлювання `a >= b` істинне тоді і тільки тоді, коли значення виразу `a` є більшим або дорівнює виразу `b`.

Приклади відношень

```
5 >= 6
x < 0
```

Операції відношення знайомі нам з математики. Проте в математиці використовувалися інші позначення, наприклад «не дорівнює» позначається «≠». Те саме стосується і логічних операцій, наприклад кон'юнкція у курсі математична логіка позначається як «&».

У цьому курсі будемо користуватися здебільшого позначеннями для арифметичних, логічних операцій, операцій відношення та інших операцій тими позначеннями, які використовуються у мові програмування Python.

Пріоритет логічних операцій та операцій відношення

Доповнимо таблицю **1.4** пріоритету операцій операціями відношення та логічними операціями.

Таблиця 2.1. Пріоритет операцій від найвищого до найнижчого

`**`
`*, /, //, %`
`+, -`
`==, !=, >, <, >=, <=`
`not`
`and`
`or`

Зауваження. Нагадаємо, що зміну пріоритету операцій можна здійснювати за допомогою дужок. Також радимо використовувати дужки у випадку, якщо є певні сумніви у тому у якому порядку виконаються операції у логічному виразі або для того, щоб зробити вираз легшим для сприйняття.

Умова – це логічний (булевий) вираз, утворений за допомогою відношень та логічних операцій, про істинність або хибність якого можна стверджувати лише у конкретний момент виконання програми.

Наприклад, умова того, що точка площини з координатами x та y належить першого координатному квадранту буде виглядати таким чином

```
x >= 0 and y >= 0
```

На відміну від звичайних висловлювань, які згідно з означенням завжди є або істинним або хибним, умова може бути невизначеною. Наприклад, вищенаведена умова невизначена, якщо не задані значення змінних `x` та `y`.

Приклад невизначеної умови при заданих значеннях змінних: умова 1/x > y невизначена при x = 0.

Тому, записуючи умови, необхідно враховувати три випадки: істинність, хибність та невизначеність.

У подальшому, якщо для деякого фіксованого стану програми умова набуває істинного значення (тобто значення **`True`**), то будемо казати, що **умова виконується**, а якщо хибного (**`False`**) – **умова не виконується**.

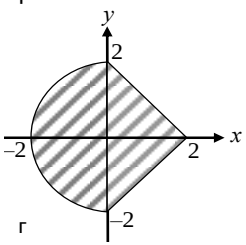
Приклад 2.1. Записати умову можливості існування трикутника із заданими сторонами a, b, c.

Розв'язок. Якщо згадати, що в будь-якому трикутнику кожна сторона менша від суми двох інших сторін, можливість існування трикутника рівносильне виконанню умови:

```
(a + b > c) and (a + c > b) and (b + c > a)
```

Звертаємо увагу на те, що дужки у цьому виразі є необов'язковими. Дійсно, арифметична операція «+» має вищий пріоритет ніж операція «>», а операція відношення «>» має вищий пріоритет ніж кон'юнкція. Проте, для того, щоб зробити умову читабельною у цьому виразі використовуються дужки.

Приклад 2.2. Записати умову приналежності точки площини з координатами (x, y) до зафарбованої області



Розв'язок. Логічні операції у геометричному контексті тісно пов'язані з операціями над множинами. Так, наприклад, якщо умови F_1 та F_2 визначають множини M_1 та M_2, то:

умова F_1 **and** F_2 визначає перетин $M_1 \cap M_2$ цих множин,

умова F_1 **or** F_2 визначає об'єднання $M_1 \cup M_2$,

умова **not** F_1 визначає доповнення до множини M_1.

Визначимо заштриховану область через операції з множинами. Заштрихована область буде об'єднанням заштрихованих підобластей з лівої $(x \le 0)$ та правої $(x > 0)$ півплощини. Отже

$$(\{x \le 0\} \cap \{x^2 + y^2 \le 4\}) \cup (\{x \ge 0\} \cap \{|x| + |y| \le 2\})$$

Звідки очевидним чином, замінивши операції над множинами логічними операціями, отримуємо умову приналежності точки до заштрихованої області.

```
((x <= 0) and (x ** 2 + y ** 2 <= 4)) or
    ((x >= 0) and (abs(x) + abs(y) < 2))
```

Логічний тип даних у Python

Реалізація алгебри висловлювань у Python представлена логічним типом даних `bool`. Змінні цього типу можуть набувати лише двох значень: **`True`**, **`False`**.

Результатом логічних операцій та умов завжди є булеві літерали **`True`** або **`False`**.

При необхідності Python автоматично зводить вираз до значення типу `bool` таким чином:

- Будь-яке число, що не дорівнює **`0`**, або непорожній об'єкт – це **`True`**.
- Число **`0`**, порожній об'єкт або значення **`None`** – це **`False`**.

Приклад 2.3. Точка площини задана декартовими координатами (x, y). Перевірити чи належить вона третьому координатному квадранту.

Розв'язок. Результатом виконання програми слід вважати значення логічного типу. При цьому будемо вважати, що значення **`True`** інтерпретується як відповідь «так» або «правда», а значення **`False`** – як відповідь «ні» або «неправда». Отже, умова буде мати вигляд:

```
x < 0 and y < 0
```

Тоді програма матиме вигляд:

```
x = float(input("x = "))
y = float(input("y = "))
check = x < 0 and y < 0  # Обчислюємо значення умови
print("Точка належить до 3-го квадранту це ", check)
```

Розгалуження

Розгалуження – це конструкція мови програмування, що забезпечує виконання визначеної команди (набору команд) у випадку виконання деякої умови або виконання однієї з кількох команд (набору команд), в залежності від значення деякої умови.

Реалізація розгалуження у Python представлена кількома керуючими конструкціями:

- умовним оператором;
- каскадним розгалуженням;
- тернарним умовним оператором.

Умовний оператор

Умовний оператор дозволяє виконувати певний набір інструкцій у випадку виконання деякої умови і інший набір інструкцій у випадку її невиконання.

Умовний оператор має такий синтаксис:

```
if condition:
    state1
else:
    state2
```

де `condition` називається умовою розгалуження, `state1` та `state2` – альтернативами.

Правила роботи умовного оператора таке: інтерпретатор обчислює значення умови `condition` і далі

- якщо умова `condition` виконується, то виконується інструкція (набір інструкцій) `state1`.
- у іншому разі, якщо умова `condition` не виконується, то виконується набір команд `state2`

Інструкції `state1` і `state2` називаються **гілками** умовного оператора.

Команди `state1` і `state2` є вкладеними інструкціями для умовного оператора. Тому, до них має застосовуватися форматування як для вкладених інструкцій – **відступ розміром 4 пробіли відносно основної інструкції.**

Приклад 2.4. Знайти модуль дійсного числа x введеного з клавіатури.

Розв'язок. Для розв'язання задачі просто скористаємося означенням модуля дійсного числа:

$$|x| = \begin{cases} x, & x \geq 0, \\ -x, & x < 0. \end{cases}$$

Таким чином програма буде мати такий вигляд:

```
x = float(input("x = "))     # Введення дійсного x
if x >= 0:
    abs_x = x
else:
    abs_x = -x
print("|", x, "|= ", abs_x) # Виведення результату
```

В умовному операторі блок **else** є необов'язковим. Тому, у ситуації, коли виконання певної інструкції (набору інструкцій) необхідно провести лише у випадку виконання деякої умови, то блок **else** опускають. При цьому такий умовний оператор називають **умовним оператором з однією гілкою**:

```
if condition:
    state
```

Виконання умовного оператора з однією гілкою буде полягати у такому: інструкція `state` виконується лише у випадку виконання умови `condition`. У зв'язку з цим, умовний оператор з однією гілкою інколи називають **захищеною інструкцією**.

Приклад 2.5. Знайти модуль дійсного числа x введеного з клавіатури.

Розв'язок. Скористаємося захищеною інструкцією у такому вигляді: якщо введене з клавіатури число є від'ємним, то поміняємо його знак на протилежний

```
x = float(input("x = "))      # Введення x
abs_x = x
if x < 0:                     # Якщо x – від'ємний
    abs_x = -x                # Міняємо знак x
print("|", x, "|= ", abs_x)   # Виведення результату
```

Приклад 2.6. Скласти програму для знаходження найбільшого з трьох значень, введених з клавіатури.

Розв'язок. Нехай $\max(\ldots)$ визначає найбільше серед чисел, що стоять у дужках. Тоді легко бачити, що $\max(x,y,z) = \max(max(x,y),z)$. Тому у програмі спочатку визначимо більше зі значень x та y, а вже потім – найбільше серед трьох чисел.

```
x = float(input("x = "))
y = float(input("y = "))
z = float(input("x = "))
# Визначаємо більше серед чисел x та y
if x < y:
    max = y
else:
    max = x
# Визначаємо більше серед чисел max(x,y) та z.
if max < z:
    max = z
print("max(", x, y, z, ")= ", max)
```

Каскадне розгалуження

Крім звичайного умовного оператора у Python є ще оператор **каскадного (множинного) розгалуження**. Каскадне розгалуження дозволяє виконати певний набір інструкцій у випадку виконання однієї серед кількох умов.

Синтаксис каскадного розгалуження:

```
if condition1:
    state1
elif condition2:
    state2
…
elif condition_N:
    state_N
else:
    state_else
```

Каскадне розгалуження виконується таким чином: інтерпретатор обчислює умову `condition1` і якщо вона виконується то інтерпретатор виконує блок інструкцій `state1`. Якщо умова `condition1` не виконується, то інтерпретатор переходить до блоку **`elif`** і обчислює умову `condition2`. Якщо умова `condition2` виконується то виконує блок інструкцій `state2`, а якщо ні, то переходить до наступного блоку **`elif`** і т.д. Якщо жодна з умов `condition1`, …, `condition_N` не є істиною, то інтерпретатор виконує блок інструкцій `state_else`.

З точки зору Python (та й будь-якої іншої мови програмування), каскадне розгалуження є послідовністю вкладених розгалужень. Саме це і породило назву для цього оператора. Отже, вищенаведене каскадне розгалуження рівносильне такому:

```
if condition1:
    state1
else:
    if condition2:
        state2
        ……
                else:
                    if condition_N:
                        state_N
                    else:
                         state_else
```

Приклад 2.7. Для введеної з клавіатури точки x знайти значення функції

$$f(x) = \begin{cases} -x^2 + 1, & x < -1, \\ 0, & |x| \le 1, \\ x^2 - 1, & x > 1. \end{cases}$$

Розв'язок. Скористаємося каскадним розгалуженням:

```
x = float(input("x = "))
if x < -1:
    f = -x ** 2 + 1
elif abs(x) <= 1:
    f = 0
else:
    f = x ** 2 - 1
print("f(", x, ")= ", f)
```

Тернарний умовний оператор

Тернарний умовний оператор (умовний вираз), це операція, що повертає одне з двох значень залежно від істинності чи хибності заданої умови. Його синтаксис такий:

```
expresion1 if condition else expresion2
```

де `condition` – умова, `expresion1` та `expresion2` – деякі вирази (значення яких можуть бути присвоєні змінній). Оператор повертає значення виразу `expresion1`, якщо умова `condition` виконується і `expresion2` у іншому разі.

Отже, згідно з визначенням тернарного оператора, інструкція

```
result = expresion1 if condition else expresion2
```

рівносильна такому розгалуженню

```
if condition:
    result = expresion1
else:
    result = expresion2
```

Використання тернарного умовного оператора дозволяє компактніше записувати програмний код

Приклад 2.8. Знайти модуль дійсного числа x введеного з клавіатури.

Розв'язок. Скористаємося тернарним умовним оператором:

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

```
abs_x = x if x >= 0 else -x
print("|", x, "|= ", abs_x)
```

Приклад 2.9. Знайти більше з двох значень, введених із клавіатури. *Розв'язок.*

```
x = float(input("x = "))
y = float(input("y = "))
max2 = x if x > y else y
print("max(", x, y, ")= ", max2)
```

Задачі для аудиторної роботи

2.1. Записати умови, що істинні тоді й тільки тоді, коли:
a) натуральне число n – парне;
b) остання цифра числа $n - 0$;
c) сума першої і другої цифри двозначного натурального числа n – двозначне число.

2.2. Точка площини задана декартовими координатами (x, y). Перевірити, чи належить вона кільцю, з центром у початку координат, внутрішнім радіусом 1 і зовнішнім 2.

2.3. Дано тризначне число. Перевірити чи
a) містить воно цифру 2;
b) складається лише з парних чисел;
c) сума його цифр дорівнює 18.

2.4. Для заданого x обчислити значення функцій

$$f(x) = \begin{cases} 0, & x \le 0, \\ x^2, & 0 < x \le 1, \\ x^4, & x > 1. \end{cases}$$

Задачі для самостійної роботи

2.5. Перевірити впорядкованість змінних a, b, c:
a) за зростанням їхніх значень;
b) за спаданням їхніх значень;
c) за зростанням чи спаданням їхніх значень.

2.6. Записати умови, що істинні тоді й тільки тоді, коли:
a) натуральне число n – непарне;
b) остання цифра натурального числа n є m;
c) ціле число n кратне натуральному числу m;
d) натуральні числа n і k одночасно кратні натуральному числу m;
e) сума цифр тризначного числа належить проміжку (a, b);

f) число x більше за число y не менше, ніж на 6;
g) принаймні одне з чисел x, y або z більше за 100;
h) тільки одне з чисел x, y або z менше за 1000.

2.7. Точка площини задана декартовими координатами (x, y). Перевірити, чи належить вона:
a) першому; b) другому; c) третьому; d) четвертому координатному квадранту.

2.8. Точка площини задана декартовими координатами (x, y). Перевірити, чи належить вона:
a) квадрату, діагоналі якого перетинаються у початку координат, а одна вершина розташована у точці $(3, 4)$;
b) внутрішності еліпса, фокуси якого розташовані на дійсній осі, велика піввісь еліпса 3, мала піввісь – 2;
c) трикутнику з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$;
d) багатокутнику з вершинами $A(x_1, y_1)$, …, $A_n(x_n, y_n)$.

2.9. Скласти програму перевірки належності точки площини (x, y) до зафарбованої області (рис. 2.1.)

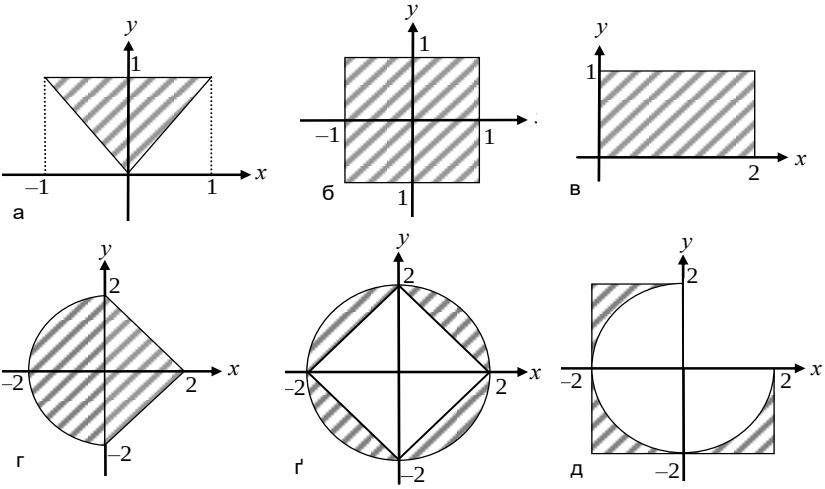


Рис. 2.1

2.10. Створити програму, яка перевіряє, чи належить початок координат трикутнику з вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$.

2.11. Точка простору задана декартовими координатами (x, y, z). Перевірити, чи належить вона
a) кулі з радіусом R і центром у початку координат.
b) циліндру, вісь якого збігається з віссю Oz, висота дорівнює h, а нижня основа лежить у площині Oxy та має радіус r.
c) зрізаному конусу, вісь якого збігається з віссю Oz, висота

дорівнює h, нижня основа лежить у площині Oxy та має радіус R, а верхня основа – радіус r.

d) внутрішності тора, що утворюється в результаті обертання круга $(x-(a+r))^2+z^2 \le r^2$навколо осі Oz.

e) тетраедру з вершинами у точках $A(x_1,y_1,z_1)$, $B(x_2,y_2,z_2)$, $C(x_3,y_3,z_3)$, $D(x_4,y_4,z_4)$.

2.12. Визначити більше та менше з двох чисел, введених з клавіатури.

2.13. Дано три дійсних числа x, y і z. Скласти програми для знаходження:

a) найбільшого за модулем; b) найменшого за модулем.

c) $\max(x+y+z, xy-xz+yz, xyz)$; d) $\max(xy, xz, yz)$.

2.14. Дано три дійсних числа x, y, z. Визначити кількість:

a) різних серед них; b) однакових серед них;

c) чисел, що є більшими за їхнє середнє арифметичне значення;

d) чисел, що є більшими за введене з клавіатури число a.

2.15. Обчислити значення функцій для заданого значення x:

a) $f(x)=\text{sign}(x)$;

b) $f(x)=\begin{cases}-x^2, & x\le -1,\\ -x^2+x^4, & |x|\le 1,\\ x^4, & x>1.\end{cases}$;

2.16. Обчислити значення функцій зображених на рис. 2.2. для заданого значення аргументу.

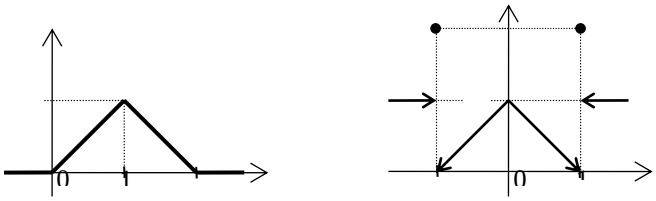


Рис. 2.2

2.17. Перевірити, чи існує трикутник із заданими сторонами a, b, c. Якщо так, то визначити, який він:

a) гострокутний; b) прямокутний; c) тупокутний.

2.18. Визначити, скільки розв'язків має рівняння та розв'язати його:

a) $ax^2+bx+c=0$; b) $ax^4+bx^2+c=0$.

2.19. Визначити, скільки розв'язків має система рівнянь і розв'язати її:

a) $\begin{cases}a_1x+b_1y+c_1=0,\\ a_2x+b_2y+c_2=0;\end{cases}$ b) $\begin{cases}|x|+|y|=1,\\ ax+by+c=0.\end{cases}$

2.20. Задано два квадрати, сторони яких паралельні координатним осям. З'ясувати, чи перетинаються вони. Якщо так, то знайти координати лівого нижнього та правого верхнього кутів прямокутника, що є їхнім перетином.

2.21. Дано два прямокутники, сторони яких паралельні координатним осям. Відомо координати лівого нижнього та правого верхнього кутів кожного з прямокутників. Знайти координати лівого нижнього та правого верхнього кутів мінімального прямокутника, що містить задані прямокутники.

2.22. Дано два прямокутники, сторони яких паралельні координатним осям. Відомо координати лівого нижнього кута кожного з прямокутників і довжини їхніх сторін. Знайти координати лівого нижнього та правого верхнього кутів мінімального прямокутника, що містить задані прямокутники.

2.23. За літерою-цифрою d визначити назву цієї цифри.

2.24. Визначити кількість днів у заданому місяці.

2.25. Для натурального числа k надрукувати фразу "Ми знайшли k грибів у лісі", узгодивши закінчення слова "гриб" з числом k.

2.26. Задано тризначне натуральне число. Записати його словами, вважаючи, що це число означає деяку суму грошей (у гривнях або іншій валюти). Наприклад, для числа 256 треба записати "двісті п’ятдесят шість гривень".

§3 ЦИКЛІЧНІ ПРОГРАМИ

Цикли, поруч з лінійними програмами та розгалуженнями, є третьою з базових конструкцій структурного програмування.

Цикл – структура керування мови програмування, що здійснює повторення певної послідовності команд.

У Python цикли поділяються на два типи:

- Цикли з умовою продовження (**`while`**).
- Цикли-ітератори по колекції (**`for`**).

Цикл з умовою продовження

Цикл з умовою продовження, це структура керування, що здійснює повторення деякої послідовності інструкцій, поки виконується задана умова.

Надалі цикл з умовою продовження будемо називати просто циклом з умовою. Синтаксис циклу з умовою такий

```
while condition:
    process_iteration
```

де `condition` називається умовою продовження циклу, `process_iteration` – тілом (або ітерацією) циклу.

Правило роботи виконання циклу з умовою:

1. Python обчислює значення умови `condition`:
2. якщо `condition == True`, то інтерпретатор виконує інструкцію `process_iteration`, після чого все починається спочатку з пункту 1.
3. якщо `condition == False`, то інтерпретатор **не** виконує інструкцію `process_iteration` і завершує виконання циклу.

Іншими словами **`while`** виконує тіло циклу, поки умова циклу є істиною. Очевидно, що якщо умова продовження циклу завжди буде залишатися істинною, то цикл один раз почавшись, ніколи не закінчить свою роботу. Така ситуація називається «зациклюванням» або «зависанням» програми. Така ситуація виникає, наприклад, якщо тіло циклу не змінює умову продовження циклу. Відповідно, програмуючи цикли, необхідно слідкувати за тим, щоб цикли були скінченними.

Приклад 3.1. Визначити чи є введене з клавіатури число N простим.

Розв'язок. Згідно з означенням, простими називаються натуральні числа більші за 1, які діляться лише на 1 і на себе. Отже, для того, щоб вияснити, чи є число простим, необхідно поділити його на всі числа від 2 до $N-1$. Якщо так станеться, що число N поділиться без остачі хоча б на одне з них, то це і буде означати, що воно не просте.

Розіб'ємо задачу на кілька кроків. Цикл, що буде перебирати всі числа від 2 до $N-1$ буде виглядати таким чином

```
i = 2              # змінна i - лічильник, починається з 2
while i <= N-1:    # поки лічильник і не перевищить N-1
    # тут буде запрограмовано алгоритм
    i = i + 1      # збільшуємо лічильник i на 1
```

Як бачимо, тут використовується змінна `i`, яка називається лічильником. Цикл буде виконуватися доки змінна-лічильник буде меншою за значення виразу `N-1`. Цикли подібні до вищенаведеного будуть досить часто зустрічатися у наших програмах.

Задача, яку ми розв'язуємо зараз, це задача пошуку, оскільки нам фактично потрібно знайти таке число яке поділить без остачі число `N`. Існує безліч алгоритмів пошуку. Розглянемо найзагальніший з них. А саме. Використаємо у нашій програмі змінну-індикатор `prime`, яка просигналізує нам, що знайдено число, на яке ділиться вихідне число `N`, тобто що воно не є простим.

```
N = int(input("N = "))
prime = True   # Вважаємо спочатку, що число є простим
i = 2          # Починаючи з 2
while i <= N-1:        # до N-1 включно
    if N % i == 0:     # якщо N ділиться без остачі на i
        prime = False  # То воно не є простим
    i = i + 1          # Беремо наступне i

if prime:
    print("Число", N, "є простим")
else:
    print("Число", N, "не є простим")
```

Наведений код програми не буде оптимальним. Дійсно, якщо на певній `i`-й ітерації при буде виявлено, що число `N` не є простим, то продовжувати цикл перевірки вже не потрібно. Для оптимізації замінимо умову цикла на таку

```
while (i <= N-1) and prime: # до N-1 включно та
                            # всі попередні числа прості
```

Приклад 3.2. Знайти суму цифр заданого натурального числа.

Розв'язок. Для розв'язання задачі треба розкласти число на цифри.

Для цього, досить згадати, що ми використовуємо позиційну десяткову систему числення. Таким чином, беручи остачу від ділення числа на 10 будемо отримувати останню цифру числа, а діленням націло на 10 прибирати цю цифру з запису числа. Наприклад,

$$256 \,\%\, 10 \;=\; 6,$$
$$256 \,//\, 10 \;=\; 25.$$

Очевидно, що повторюючи ці дві операції поки число лишається більшим за нуль і підсумовуючи у деякій змінній отримані остачі отримаємо бажаний результат. Програма буде мати вигляд

```
N = int(input("N = "))
suma = 0              # Змінна, у якій буде міститися сума
while N > 0:          # Поки число не нульове (має цифри)
    last = N % 10     # Визначаємо останню цифру числа
    suma += last      # Додаємо її до змінної suma
    N //= 10          # Прибираємо останню цифру числа
print("Сума цифр заданого числа є", suma)
```

Приклад 3.3. Дано натуральне число K. Скласти програму знаходження всіх натуральних чисел менших або рівних K, які при піднесенні до квадрата дають паліндром.

Розв'язок. Натуральне число називається паліндромом, якщо його запис читається однаково зліва направо і справа наліво. Наприклад, числа

$$12321, 626, 10001$$

є паліндромами.

Для чисел $k = 1, \ldots, K$ будемо порівнювати значення $N = k^2$ із числом P, що є записом числа N у зворотному порядку.

Розіб'ємо задачу на два етапи. Для початку напишемо програму, яка будує число P, що є записом числа N у зворотному порядку. Ця програма, майже повністю повторює програму з попереднього прикладу. Єдина відмінність полягає у тому, що будемо шукати не суму числа, а будувати число P з огляду на позиційну десяткову систему числення:

$$625 = (62) * 10 + 5 = \big((6) * 10 + \ 2\big) * 10 + 5$$

Отже вихідний код, для обернення числа буде виглядати таким чином

```
# Вважаємо, що змінна N задана
P = 0      # P буде містити обернений запис числа N
while N > 0:
    last = N % 10
    P = P * 10 + last
    N //= 10
```

Після виконання цього коду, у змінній `P` буде міститися число `N` записане у зворотному порядку.
Нарешті напишемо програму, що розв'язує поставлену задачу.

```
K = int(input("K = "))
k = 1
while k <= K:
    N = k * k

    # Будуємо обернений запис числа N
    P = 0  # P буде містити обернений запис числа N
    while N > 0:
        last = N % 10
        P = P * 10 + last
        N //= 10

    # Якщо обернений запис числа N збігається з N=k**2
    if P == k * k:
        print(k) # Виводимо k на екран
    k += 1
```

Приклад 3.4. Написати програму для обчислення найбільшого спільного дільника двох цілих чисел за допомогою алгоритма Евкліда.

Розв'язок. Для розв'язання задачі скористаємося модифікованим алгоритмом Евкліда, у якому послідовне віднімання замінюється знаходженням остачі від ділення.

```
N = int(input("Введіть перше число "))
M = int(input("Введіть друге число "))

# Запам'ятовуємо вхідні змінні
U = N
V = M
# Модифікуємо змінні U та V так, щоб в
# U було більше число, а у V - менше
```

```
if U < V:
    P = U
    U = V
    V = P
# Запускаємо алгорим Евкліда
while V > 0:
    P = V
    V = U % V
    U = P
# Виводимо результат на екран
print("НСД(%d, %d) = %d" % (N, M, U))
```

Цикл по колекції.

Колекція – це структура, що містить скінченний набір елементів, до яких реалізовано доступ.

Колекції розрізняють в залежності від типу елементів у структурі, способів зображення її елементів у пам'яті, а також доступу до елементів.

Прикладом колекції може бути послідовність

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Ітератор – це інтерфейс, що надає послідовний доступ до елементів колекції та навігацію по ним.

Для перебору всіх елементів колекції у Python використовується цикл по колекції елементів.

Цикл по колекції – це структура керування, що здійснює повторення деякої послідовності інструкцій, для всіх елементів колекції.

Через призначення, яке виконують цикли по колекції їх часто називають «для всіх» (eng. "for each").

Синтаксис циклу по колекції такий

```
for iterator in collection:
    process_iteration
```

змінна `iterator` це ітератор, який послідовно проходить всі елементи колекції `collection`. `process_iteration` – тіло циклу, що виконується для кожного поточного значення змінної `iterator` з колекції.

Розглянемо приклад використання циклу по колекції, котра є послідовністю натуральних чисел від 1 до 9. Цикл по колекції у цьому прикладі використовується для того, щоб підрахувати суму чисел колекції.

```
suma = 0
collection = {1, 2, 3, 4, 5, 6, 7, 8, 9} # колекція
for i in  collection :                   # i – ітератор
    suma += i
print(suma)
```

Однією з найпростіших, проте дуже важливих колекцій у Python є арифметична прогресія натуральних чисел

$$a_0 = b,\ a_1 = a_0 + d, \dots, a_n = a_{n-1} + d$$

Наприклад, якщо $b = 1, d = 2$, то колекція, найбільший член якої менший за число $c = 10$, буде

$$\{1,3,5,7,9\}$$

У Python для побудови такої колекції існує спеціальна функція

```
range(b,c,d)
```

вона повертає впорядковану колекцію, що містить арифметичну прогресію, перший член якої `b`, останній член менший за `c`, а різниця прогресії дорівнює `d`. Наприклад, для створення колекції, що містить непарні числа, від 3 до 11 треба викликати функцію

```
range(3,12,2)
```

Якщо різниця прогресії $d = 1$, то `d` можна опускати:

```
range(b,c)
```

Крім того, якщо перший член прогресії $b = 0$, то `b` можна також опускати:

```
range(c)
```

Отже, попередній приклад, підрахунку суми натуральних чисел від 1 до 9 можемо переписати у такий спосіб:

```
suma = 0
for i in range(1, 10):
    suma += i

print(suma)
```

Приклад 3.5. Знайти суму парних чисел з діапазону від 1 до N.

Розв'язок. Для розв'язання задачі треба згенерувати колекцію парних чисел, після чого просумувати її члени

Інструкція `range(2, N + 1, 2)` створює колекцію парних чисел з діапазону [2, N]. Таким чином, для розв'язання задачі досить підсумувати члени створеної колекції.

```
N = int(input("N = "))
suma = 0
for i in range(2, N + 1, 2):
    suma += i
print("Сума парних чисел від 1 до %d є %d" % (N, suma))
```

Цикл **for** у комплексі з інструкцією `range` часто використовується для того, щоб виконати певну інструкцію наперед задану кількість разів. Наприклад, інструкція

```
for i in range(10):
    process_iteration
```

виконає тіло циклу `process_iteration` рівно 10 разів, оскільки ітератор циклу (який у цьому випадку буде лічильником) `i` пробігає значення з діапазону від 0 до 9.

Приклад 3.6. Скласти програму для обчислення добутку двох натуральних чисел m і n, використовуючи тільки операцію додавання.

Розв'язок. Легко помітити, що

$$m \cdot n = \underbrace{n + \ldots + n}_{m \text{ разів}}$$

Таким чином, програма має вигляд

```
m = int(input("m = "))
n = int(input("n = "))
s = 0
for i in range(m):
```

```
    s += n
print(s)
```

Приклад 3.7. Дано натуральне число n. Написати програму для обчислення значень многочлена

$$y(x) = x^n + x^{n-1} + \ldots + x^2 + x + 1$$

при заданому значенні x.

Розв'язок. Розглянемо два способи розв`язання.

Спосіб 1. Позначимо $z_k = x^k, k \geq 0$. Тоді фрагмент вихідного коду

```
z = 1
for i in range(1, n + 1):
    z *= x
```

забезпечить послідовне обчислення у змінній `z` значень $z_0, z_1, \ldots, z_n$. Таким чином, отримаємо програму

```
x = int(input("x = "))
n = int(input("n = "))

z = 1
y = 1
for i in range(1, n + 1):
    z *= x
    y += z

print("y(%f) = %f" %(x, y))
```

Спосіб 2. Скористаємось схемою Горнера. Розставивши дужки таким чином:

$$y = x^n + x^{n-1} + \cdots + x^2 + x + 1 = = (\ldots(x + 1)x + 1)x + \ldots + 1)x + 1$$

отримаємо програму

```
x = int(input("x = "))
n = int(input("n = "))

y = 1
for i in range(1, n + 1):
    y = y * x + 1
```

```
print("y(%f) = %f" %(x, y))
```

Приклад 3.8. Нехай задано натуральне число $n \geq 1$ і послідовність чисел $a_1, \dots, a_n$. Знайти $\max(a_1, \dots, a_n)$.

Розв'язок. Для розв'язання задачі досить помітити, що

$$max(a_1, a_2, \dots, a_n) = max(\dots(max(max(a_1, a_2), a_3), \dots), a_n).$$

Нехай у змінній `a` міститься чергове число послідовності a_n, що вводиться з клавіатури. Оскільки кількість членів послідовності a_n відома, то для їхнього зчитування використаємо цикл по проміжку значень. Для визначення максимального числа використаємо змінну `max`. Таким чином, програма буде мати вигляд:

```
N = int(input("N = "))
# Зчитуємо 1-й член послідовності
a = float(input("Задайте член послідовності = "))

max = a                 # Вважаємо, що він є найбільшим
for i in range(1, N): # Цикл виконається рівно N-1 раз
    a = float(input("Уведіть член послідовності"))
    # У змінну max записуємо більше з числом max і
    # поточного члена зчитаного з клавіатури
    max = a if a > max else max

print("Найбільшим є число", max)
```

Приклад 3.9. Написати програму для обчислення подвійного факторіала натурального числа n : $y = n!!$

Розв'язок. За означенням

$$n!! = \begin{cases} 1 \cdot 3 \cdot 5 \cdot \ldots \cdot n, & n - \text{непарне}, \\ 2 \cdot 4 \cdot 6 \cdot \ldots \cdot n, & n - \text{парне}. \end{cases}$$

В обох випадках маємо як співмножники всі члени спадної арифметичної прогресії з різницею -2, які містяться між n та 1. Звідси програма

```
n = int(input("n = "))

p = 1
for k in range(n, 0, -2):
    p = p * k

print("%d!! = %d" % (n, p))
```

розв`язує задачу. Зауважимо, що при непарному n останнім значенням k буде 1, а при парному буде 2.

Переривання та продовження циклів

Для переривання виконання циклу (як **while** так і **for**) або ігнорування частини тіла циклу, у Python існують дві спеціальні команди **break** та **continue**.

Оператор **break** достроково перериває виконання циклу. Оператор **continue** починає наступний прохід циклу, минаючи невиконаний залишок тіла циклу.

Приклад 3.10. Задано послідовність чисел, що завершується числом 0, серед елементів якої є принаймні одне додатне число. Потрібно знайти найбільше серед додатних чисел.

Розв'язок. Зчитувати члени послідовності будемо у циклі. При цьому, якщо зчитане число дорівнює нулю, то будемо використовувати **break** для завершення виконання циклу. Якщо число менше нуля, то будемо використовувати **continue**, щоб пропустити це число і не враховувати його у обчисленні найбільшого.

```
max = 0     # Поточний найбільший член послідовності
while True:         # Завершення циклу буде через break
    a = float(input("Задайте член послідовності"))
    if a == 0:      # якщо введене число 0,
        break       # завершуємо цикл
    elif a < 0:     # якщо введене число < 0,
        continue    # переходимо на початок циклу
    max = a if a > max else max

print("Найбільшим є число", max)
```

До операторів циклів **while** та **for** може застосовуватися оператор **else**. Синтаксис його використання для циклу **while** такий

```
while condition:
    process_iteration
else:
    state_else
```

а для циклу **for** такий

```
for iterator in collection
```

```
    process_iteration
else:
    state_else
```

Оператор **else** перевіряє чи відбувся вихід з циклу за допомогою оператора **break**. Блок інструкцій state_else виконується лише у тому випадку, якщо вихід з циклу відбувся без допомоги оператора **else**.

Приклад 3.11. Задано послідовність чисел. Перевірити чи міститься серед них число a.

Розв'язок. Для зчитування членів послідовності скористаємося циклом **for**. При цьому, якщо зчитане число дорівнює заданому числу a, то виведемо на екран відповідне повідомлення та припинимо виконання циклу інструкцією **break**. Якщо числа a серед членів послідовності не знайдено, то цикл завершиться природнім чином і у блоці **else** можемо вивести повідомлення, що числа a серед членів послідовності не знайдено. Отже, програма буде мати вигляд

```
N = int(input("Задайте кількість членів послідовності "))
a = int(input("Задайте шукане число "))

for i in range(N):
    current = float(input("Задайте поточний елемент "))
    if current == a:
        print("Послідовність містить число ", a)
        break
else:
    print("Послідовність не містить число ", a)
```

Рекурентні співвідношення

Рекурентне співвідношення першого порядку

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

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

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

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

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

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

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

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

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

Нехай послідовність $\mathrm{a_n}$ задана рекурентним співвідношенням

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

Тоді, після виконання коду

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

у змінній `a` буде міститися значення елемента a_N послідовності

Приклад 3.12. Для введеного з клавіатури значення N обчислити $N!$

Розв'язок. Як було зазначено раніше послідовність $a_n = n!$ може бути задана рекурентним співвідношенням

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

Тоді, згідно з вищенаведеним алгоритмом, отримаємо

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

Приклад 3.13. Скласти програму для обчислення елементів послідовності

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

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

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

Звідки випливає, що для $n \geq 1$

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

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

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

Отже остаточно отримаємо рекурентне співвідношення першого порядку

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

Тоді, згідно з вищенаведеним алгоритмом, отримаємо програму

```
N = int(input("N = "))
x = float(input("x = "))
a = 1
for n in range(1, N+1):
    a = x / n * a          # можна так: a *= x / n
print ("a = ", a)          # виводимо на екран результат
```

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

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

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

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

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

Приклад 3.14. Скласти програму обчислення суми:

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

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

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

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

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

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

```
N = int(input("N = "))
S = 1
for n in range(2, N + 1):
    S += 1 / n
print("S = ", S)
```

Приклад 3.15. Скласти програму обчислення суми

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

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

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

Отже, рекурентне співвідношення для буде мати вигляд

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

і відповідно програма

```
N = int(input("N = "))
S = 1
for n in range(2, N + 1):
    S = 2 * S + n ** 2
print("S = ", S)
```

Рекурентні співвідношення старших порядків

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

Тоді

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

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

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

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

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

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

$$\begin{gathered} F_0 = 1,\ F_1 = 1 \\ F_2 = F_1 + F_0 = 1 + 1 = 2 \\ F_3 = F_2 + F_1 = 2 + 1 = 3 \\ F_4 = F_3 + F_2 = 3 + 2 = 5 \\ F_5 = F_4 + F_3 = 5 + 3 = 8 \\ F_6 = F_5 + F_4 = 5 + 3 = 13 \end{gathered}$$

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

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

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

Тоді, після виконання коду

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

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

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

Приклад 3.16. Знайти N-й член послідовності Фібоначі

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

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

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

```
N = int(input("N = "))
F2 = 1
F1 = 1
for n in range(2, N + 1):
    F = F1 + F2
    F2 = F1
    F1 = F
print("F = ", F)
```

Приклад 3.17. Скласти програму для обчислення визначника порядку n:

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

Розв'язок. Легко обчислити, що

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

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

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

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

```
N = int(input("N = "))
D2 = 5    # 1-й член послідовності
```

```
D1 = 19  # 2-й член послідовності
for n in range(3, N + 1):
    D = 5 * D1 - 6 * D2
    D2 = D1
    D1 = D
print("D_%d = %d" % (N, D1))
```

Системи рекурентних співвідношень

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

Розглянемо системи рекурентних співвідношень на прикладах

Приклад 3.18. Скласти програму для обчислення N-го члена послідовності, що визначається сумою

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

Розв'язок. Розкриваючи суму побачимо, що для всіх $n \geq 1$

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

Отже послідовність S_n визначається рекурентним співвідношенням

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

Позначимо

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

У прикладі 3.13 для цієї послідовності було отримано рекурентне співвідношення. Тоді для вихідної послідовності S_n система рекурентних співвідношень матиме вигляд

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

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

```
N = int(input("N = "))
x = float(input("x = "))
a = 1
S = 1
for n in range(1, N+1):
    a = x / n * a
    S = S + a
print ("S = ", S)
```

Приклад 3.19. Скласти програму для обчислення суми

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

Розв'язок. Рекурентне співвідношення можемо побудувати двома способами.

Спосіб 1. Очевидно, що $S_0 = 1$. Розкриваючи суму і групуючи доданки аналогічно до прикладу 3.15, отримаємо

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

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

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

Таким чином, отримаємо систему рекурентних співвідношень

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

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

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

Тоді система рекурентних співвідношень буде мати вигляд

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

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

```
N = int(input("N = "))
a = float(input("a = "))
b = float(input("b = "))

S = x = 1
for n in range(1, N + 1):
    x = a * x
    S = b * S + x

print(S)
```

Приклад 3.20. Обчислити суму, задану рекурентним співвідношенням

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

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

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

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

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

```
print(S)
```

Приклад 3.21. Обчислити суму, задану рекурентним співвідношенням

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

де

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

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

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

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

S = 0.5
a = 1
b = 1
for n in range (1, N+1):
    a_k = a * b   # допоміжна змінна для a_k
    b_k = a + b   # допоміжна змінна для b_k
    a = a_k
    b = b_k
    S = S + a / (1 + b)

print(S)
```

Приклад 3.22. Обчислити добуток, заданий рекурентним співвідношенням

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

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

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

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

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

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

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

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

P = 1.0 / 9.0
z = 9
t = 2
a2 = a3 = 1
a1 = 3
for n in range(3, N + 1):
    z = z * 3
    t = t * 2
    a = a3 + a2 / t
    a3 = a2
    a2 = a1
    a1 = a
    P = P * a / z

print(P)
```

Відшукання членів послідовності, що задовольняють умову

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

Розглянемо приклади

Приклад 3.23. Для довільного натурального $N \geq 2$ знайти найменше число вигляду 3^k, де k – натуральне, таке, що $3^k \geq N$.

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

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

Отже, враховуючи, що послідовність a_k строго зростаюча, щоб виконати завдання задачі необхідно обчислювати члени послідовності a_k в циклі використовуючи вищенаведене рекурентне співвідношення, доки не знайдемо перший такий, що $a_k \geq N$. Відповідно умова у циклі, буде запереченням до $a_k \geq N$, тобто $a_k < N$. Далі очевидним чином маємо програму

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

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

print(a)
```

Приклад 3.24. Послідовність задана рекурентним співвідношенням

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

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

Розв'язок. Запишемо кілька перших членів заданої послідовності

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

Звернемо увагу на те, що ця послідовність є зростаючою. Тоді, для того, щоб знайти найбільший член цієї послідовності, що не перевищує задане число a, необхідно обчислювати члени цієї послідовності, доки не знайдемо перший такий член, що буде більшим за задане число a. Тоді, член послідовності, що вимагається умовою задачі – елемент послідовності, що передує знайденому. Наприклад, якщо $a = 30$, то перший член послідовності, що більший за число 30 є 41, а відповідно шуканий згідно з умовою задачі член послідовності – той який йому передує, тобто 19.

Нехай `x, x1, x2, x3` – змінні, що використовуються згідно до алгоритму для обчислення членів послідовності заданої вищенаведеним рекурентним співвідношенням третього порядку. Нагадаємо, що тоді у змінних `x, x1` будуть знаходитися поточні члени послідовності x_n. Тоді, умова циклу буде `x1 <= a`.

Таким чином, маємо програму

```
a = int(input("a = "))
x3 = 1
x2 = 0
x1 = 1
n = 2              # Номер поточного члену послідовності
```

```
while x1 <= a:    # Поки поточний член послідовності
    n += 1
    x = x1 + 2 * x2 + x3
    x3 = x2
    x2 = x1
    x1 = x

print ("x(%d) = %d <= %d = a" % (n - 1 , x2, a))
```

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

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

Наближені обчислення границь послідовностей

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

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

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

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

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

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

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

Розглянемо застосування зазначеного підходу на прикладах.

Приклад 3.25. Скласти програму наближеного обчислення золотого перетину c, використовуючи:

а) границю

$$\Phi = \lim_{n\to\infty} \frac{F_n}{F_{n-1}}$$

де F_n – послідовність чисел Фібоначчі;

б) ланцюговий дріб

$$\Phi = 1 + \frac{1}{1 + \frac{1}{1+\ddots}}.$$

Розв'язок. Золотий перетин це число

$$\Phi \approx 1{,}6180339887..$$

яке має багато унікальних властивостей, причому не лише математичних. Це число можна зустріти як у різноманітних сферах діяльності людини (наприклад, у мистецтві, архітектурі, культурі) так і у оточуючому нас світі, зокрема фізиці, біології тощо. Для того, щоб переконатися у тому, що наш алгоритм працює правильно, скористаємося однією з властивостей золотого перетину, а саме:

$$\Phi - 1 = \frac{1}{\Phi}.$$

а) Розглянемо послідовність

$$\Phi_n = \frac{F_n}{F_{n-1}}, n \geq 1.$$

Знайдемо рекурентне співвідношення для c_n. Очевидно, що $c_1 = 1$, далі для $n \geq 2$ отримаємо

$$\Phi_n = \frac{F_n}{F_{n-1}} = \frac{F_{n-1} + F_{n-2}}{F_{n-1}} = 1 + \frac{1}{\frac{F_{n-1}}{F_{n-2}}} = 1 + \frac{1}{\Phi_{n-1}}.$$

б) Розглянемо послідовність

$$\Phi_n = 1 + \frac{1}{1 + \frac{1}{1+\ddots + \frac{1}{1}}},$$

що містить $n-1$ риску дробу. Очевидно, що для цієї послідовності рекурентне співвідношення буде таким же, як у пункті а).

Напишемо програму, що знаходить наближене з точністю ε значення границі послідовності Φ_n. Використаємо змінну `current` для обчислення поточного члену послідовності Φ_n і змінну `prev`, у якій будемо запам'ятовувати попередній член Φ_{n-1} цієї послідовності.

Тоді програма має вигляд

```
eps = 0.0000000001  # точність
prev = 0    # попередній член послідовності
current = 1 # поточний член послідовності

while abs(current - prev) >= eps:
    prev = current
    current = 1 + 1 / current

print("Φ =", current)  # Виводимо значення золотого перетину

# Перевірка результату згідно з властивостями
print("Φ - 1 =", current - 1)
print("1 / Φ =", 1 / current)
```

Наведемо результат виконання програми.

```
Φ = 1.618033988738303
Φ - 1 = 0.6180339887383031
1 / Φ = 0.6180339887543225
```

Приклад 3.26. За допомогою розкладу функції e^x в ряд Тейлора

$$y(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

обчислити з точністю $\varepsilon > 0$ її значення для заданого значення x.

Розв'язок. Позначимо загальний член вищенаведеного ряду через a_n, а його часткову суму

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

Очевидно, що $S_n \to e^x$, $n \to \infty$.

У прикладі 3.18 було отримано, що послідовність S_n визначається системою рекурентних співвідношень

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

Відповідно до означення, наведеного вище і з огляду на рекурентне співвідношення, під наближеним значенням границі послідовності S_n будемо розуміти такий член S_N , що виконується співвідношення

$$|S_N - S_{N-1}| = |a_N| < \varepsilon$$

Отже, програма буде мати вигляд

```
eps = 0.0000000001  # точність

x = float(input("x = "))
a = 1
S = 1
n = 0
while abs(a) >= eps:
    n += 1
    a = x / n * a
    S = S + a

print("exp(%f) = %f" % (x, S))
```

Звернемо увагу на те, що на відміну від цикла **for**, цикл **while** не має вбудованого лічильника. Тому, оскільки нам необхідно враховувати у формулі номер члена послідовності, то ми задали змінну n, яка відіграє роль лічильника.

Наведемо результат виконання програми.

```
x = 1.0
exp(1.000000) = 2.718282
```

Найпростіші методи розв'язання алгебраїчних рівнянь

Розглянемо алгебраїчне рівняння вигляду

$$f(x) = 0,$$

де функція $f(x)$ неперервна на відрізку $[a, b]$.

Розглянемо найпростіші наближені методи відшукання розв’язків рівнянь вищенаведеного вигляду на заданому відрізку $[a, b]$. До таких методів належать **метод бісекції** та **метод хорд**.

Метод бісекції або метод ділення відрізка навпіл, полягає у такому. Припустимо, що відомо, що на відрізку $[a, b]$ існує єдиний корінь цього рівняння. Тоді очевидно, що на кінцях цього відрізка функція набуває різних знаків. Поділивши відрізок навпіл, будемо брати ту з половин, для якої на кінцях функція буде набувати значення різних знаків. Якщо середина відрізка

виявиться нулем функції, то процес завершується. Якщо задана точність обчислення ε, то вищеописану процедуру поділу слід виконувати доти, доки довжина відрізка не стане меншою за ε, а наближеним значенням тоді вважають середину цього відрізка.

Метод хорд полягає в обчислені елементів послідовності $\{u_n : n \geq 0\}$, визначеної рекурентним співвідношенням

$$u_0 = a;$$
$$u_n = u_{n-1} - f(u_{n-1}) \cdot \frac{b - u_{n-1}}{f(b) - f(u_{n-1})}, \qquad n \geq 1$$

до виконання умови $|u_n - u_{n-1}| < \varepsilon$.

Приклад 3.27. Використовуючи метод бісекції знайти корінь рівняння $\tan x = 2x$ на відрізку $[0.5, 1.5]$ із заданою точністю ε.

Розв'язок. Згідно з вищенаведеним алгоритмом, для розв'язання рівняння розглянемо допоміжну функцію вигляду

$$f(x) = tan\, x - 2x.$$

Тоді програма буде мати вигляд

```
import math                 # підключення математичної бібліотеки
eps = 0.0000000000001       # точність
l = 0.5                     # лівий кінець відрізка
r = 1.5                     # правий кінець відрізка
# tan(l) - 2 * l < 0, tan(r) - 2 * r > 0

while r - l >= eps:
    x = (l + r) / 2.0  # середина відрізка [l,r]
    f = math.tan(x) - 2.0 * x
    if f == 0.0:
        break
    elif f < 0.0:
        l = x  # [l,r] = [x,r]
    else:
        r = x  # [l,r] = [l,x]

# корінь – середина останнього відрізка [l,r]
x = (l + r) / 2.0
print(x)
```

Задачі для самостійної роботи

3.1. Вивести на екран такий рядок:

```
n! = 1*2*3*4*5*...*n
```

де n – введене з клавіатури число.

3.2. Вивести на екран таблицю множення на 5:

```
1 x 5 = 5
2 x 5 = 10
...
9 x 5 = 45
```

3.3. Вивести на екран таблицю:

```
 1  |  2  |  3  |...|  n-1  |  n
----------------...-------------
a+1 | a+2 | a+3 |...| a+n-1 | a+n
```

де a, n – вводяться з клавіатури.

3.4. Написати програму друку таблиці значень функції $y = \sin(x)$ на відрізку $[0; 1]$ з кроком $h = 0.1$.

3.5. Написати програму для обчислення добутку двох натуральних чисел, використовуючи лише операцію додавання.

3.6. Написати програму для обчислення натурального степеня n від дійсного числа a, використовуючи лише операцію множення.

3.7. Написати програму для обчислення

a) факторіала натурального числа $n!$;

b) подвійного факторіала натурального числа $n!!$.

3.8. Написати програми для обчислення факторіалів:

a) $y = (2n)!!$ b) $y = (2n + 1)!!$ c) $y = n!\, n!!\, (n + 1)!!$

3.9. Задані дійсні числа x, y $(x > 0, y > 1)$. Скласти програму для знаходження цілого числа k (додатного, від'ємного або рівного нулю), що задовольняє умову $y^{k-1} \le x < y^k$.

3.10. Задано натуральне число n і цілі числа $a, a_1, \ldots, a_n$.

a) Знайти номер члена послідовності $a_1, \ldots, a_n$, який дорівнює a. Якщо такого члена послідовності немає, то результатом має бути число 0;

b) Якщо в послідовності $a_1, \ldots, a_n$ є хоча б один член, що дорівнює a, то отримати суму всіх членів, які йдуть за таким членом послідовності. У протилежному випадку відповіддю має бути відповідне текстове повідомлення.

c) Отримати суму додатних, кількість від'ємних і число нульових членів послідовності $a_1, \ldots, a_n$.

d) Визначити, скільки серед членів послідовності більших за своїх "сусідів", тобто за попереднє та наступне число.

3.11. Задано натуральні числа n і p та цілі числа $a_1, \ldots, a_n$. Скласти програму обчислення добутку членів послідовності, кратних p.

3.12. Задані натуральне число n, дійсні числа $a_1, a_2, \ldots, a_n$. Написати програми для знаходження:

a) $\min(a_1, a_2, \dots, a_n)$;
b) $\max(|a_1|, \dots, |a_n|)$;
c) $\max(a_2, a_4, \dots)$;
d) $\min(a_1, a_3, \dots)$;
e) $\min(a_2, a_4, \dots) + \max(a_1, a_3, \dots)$;
f) $\max(a_1, a_2, a_4, a_8, \dots)$;
g) $\max(-a_1, a_2, -a_3, \dots, (-1)^n a_n)$;
h) $\max(a_1, 2a_2, \dots, na_n)$;
i) $(\min(a_1, \dots, a_n))^2 - \min(a_1^2, \dots, a_n^2)$;
j) $\max(a_1 + a_2, \dots, a_{n-1} + a_n)$.
k) того з них, яке є найближчим до цілого числа.

3.13. Задані натуральне число n, натуральні числа $a_1, a_2, \dots, a_n$. Написати програми для знаходження:

a) кількості парних серед $a_1, a_2, \dots, a_n$;
b) кількості повних квадратів серед $a_1, a_2, \dots, a_n$;
c) кількості квадратів непарних чисел серед $a_1, a_2, \dots, a_n$.

3.14. Задані натуральне число n, дійсні числа $a_1, a_2, \dots, a_n$. Скласти програму, що визначає в послідовності $a_1, a_2, \dots, a_n$ кількість сусідств:

a) двох додатних чисел;
b) двох чисел різних знаків.

3.15. Дана непорожня послідовність різних натуральних чисел, за якою йде 0. Визначити порядковий номер найменшого з них.

3.16. Дана непорожня послідовність різних дійсних чисел, серед яких є хоча б одне від`ємне число, за якою йде 0. Визначити величину найбільшого серед від`ємних членів цієї послідовності.

3.17. Задана послідовність дійсних чисел $x_1, x_2, \dots, x_n$ ($n \geq 3$ заздалегідь невідоме), за якою йде 0. Скласти програми для обчислення:

a) $nx_1 + (n-1)x_2 + \dots + 2x_{n-1} + x_n$;
a) $(x_1 + 2x_2 + x_3) \cdot (x_2 + 2x_3 + x_4) \cdot \dots \cdot (x_{n-2} + 2x_{n-1} + x_n)$;
b) $(x_1 + x_2 + x_3) \cdot x_2 + \dots + (x_{n-2} + x_{n-1} + x_n) \cdot x_{n-1}$.

3.18. Написати програму визначення кількості тризначних натуральних чисел, сума цифр яких дорівнює n ($n \geq 1$). У програмі не використовувати операцію ділення.

3.19. Створити програму, яка з'ясовує, чи входить задана цифра до запису заданого натурального числа.

3.20. Використовуючи формулу

$$\text{НСК}(m, n) = \frac{m \cdot n}{\text{НСД}(m, n)}$$

де $\text{НСД}(m, n)$ – найбільших спільний дільник чисел m і n, скласти програму для обчислення найменшого спільного кратного двох натуральних чисел m і n.

3.21. Дано натуральні числа m і n. Скласти програму знаходження всіх їх спільних дільників.

3.22. Дано натуральні числа m і n. Написати програму для знаходження всіх їх натуральних спільних кратних, менших за $m \cdot n$.

3.23. Дано натуральні числа m і n. Знайти такі натуральні числа p і q, не виключаючи спільних дільників, що $p/q = m/n$.

3.24. Знайти всі прості нескоротні дроби, що містяться між 0 і 1, знаменники яких не перевищують 7 (дріб задається двома числами –

чисельником та знаменником).

3.25. Дано натуральне число n. Знайти всі такі натуральні q, що n ділиться на q^2 і не ділиться на q^3.

3.26. Два натуральних числа називаються "дружніми", якщо кожне з них дорівнює сумі всіх дільників іншого, крім самого цього числа. Скласти програму знаходження всіх пар "дружніх" чисел, що лежать у діапазоні $[200,300]$.

3.27. Дано натуральне число n. Скласти програму знаходження всіх Піфагорових трійок натуральних чисел, кожне з яких не перевищує n, тобто всіх таких трійок натуральних чисел a, b, c, що $a^2 + b^2 = c^2 \quad (a \le b \le c \le n)$.

3.28. Натуральне число із n цифр є числом Армстронга, якщо сума його цифр, піднесених до n-го степеня, дорівнює самому числу (наприклад, $153 = 1^3 + 5^3 + 3^3$).Скласти програму знаходження всіх чисел Армстронга, що складаються з двох, трьох та чотирьох цифр.

3.29. Дано натуральне число n. Скласти програму, що визначає, чи можна подати його у вигляді суми двох квадратів натуральних чисел. Якщо це можливо, то

a) вказати пару a, b таких натуральних чисел , що $n = a^2 + b^2$;
b) вказати пару a, b таких натуральних чисел, що $n = a^2 + b^2$, $a \ge b$.

3.30. Дано натуральне число n. Скласти програму, що дозволяє

a) переставити першу та останню цифри числа n;
b) приписати по одиниці до початку та кінця запису числа n;
c) одержати суму m останніх цифр числа n.

3.31. Дано натуральне число n. Скласти програму, що визначає серед чисел $1, \ldots, n$ усі такі, запис яких збігається з останніми цифрами запису їхніх кубів (наприклад, $4^3 = 64$, $25^3 = 15625$).

3.32. Натуральне число називається паліндромом, якщо його запис читається однаково зліва направо і справа наліво (наприклад, 1, 393, 4884). Скласти програму, що визначає, чи є задане натуральне число n паліндромом.

3.33. Маємо ціле $n > 2$. Скласти програму для знаходження всіх простих чисел із діапазону $[2, n]$, які є

a) числами послідовності Фібоначчі;
b) паліндромами;
c) числами виду $q^2 + 1$, де q – ціле число.

3.34. Скласти програму знаходження всіх простих дільників заданого натурального числа.

3.35. Число називається досконалим, якщо воно дорівнює сумі всіх своїх дільників, крім самого цього числа (наприклад, числа 6 і 28: $6 = 1 + 2 + 3$, $28 = 1 + 2 + 4 + 7 + 14$). Написати програму, яка визначає, чи є задане натуральне число n досконалим.

3.36. Перевірити, чи існують досконалі числа з проміжку $[2, n]$, котрі є повними квадратами. Якщо так, то вивести на екран перше з них.

3.37. Дано натуральне число n. Визначити, чи можна подати $n!$ у вигляді добутку трьох послідовних чисел.

3.38. Дано натуральне число n. Написати програми для обчислення значень виразів:

a) $\sqrt{2+\sqrt{2+\cdots+\sqrt{2}}}$ (n коренів);

b) $\sqrt{3+\sqrt{6+\cdots+\sqrt{3(n-1)+\sqrt{3n}}}}$;

3.39. Дано натуральне число n. Написати програми для обчислення значень виразів при заданому значенні x:

a) $x^n + x^{n-1} + \ldots + x^2 + x + 1$;

b) $1 + (x-1) + (x-1)^2 + \ldots + (x-1)^n$;

c) $y = x^{2^n} + x^{2^{n-1}} + \ldots + x^4 + x^2 + 1$,

d) $(nx)^n + ((n-1)x)^{n-1} + \ldots + (2x)^2 + x$.

e) $1 + \frac{1}{x^2+1} + \frac{1}{(x^2+1)^2} + \ldots + \frac{1}{(x^2+1)^n}$;

f) $1 + \sin x + \ldots + \sin^n x$.

g) $\cos\pi + \cos\frac{\pi}{2} + \cos\frac{\pi}{4} + \ldots + \cos\frac{\pi}{2^n}$.

3.40. Створити програми для обчислення елементів послідовностей:

a) $x_k = \frac{x^k}{k}$ $(k \geq 1)$

b) $x_k = \frac{(-1)^k x^k}{k}$ $(k \geq 1)$

c) $x_k = \frac{x^k}{k!}$ $(k \geq 0)$

d) $x_k = \frac{(-1)^k x^k}{(k^2+k)!}$, $k \geq 0$;

e) $x_k = \frac{x^{2k}}{(2k)!}$ $(k \geq 0)$

f) $x_k = \frac{(-1)^k x^{2k}}{(2k)!}$ $(k \geq 0)$

g) $x_k = \frac{x^{2k+1}}{(2k+1)!}$ $(k \geq 0)$

h) $x_k = \frac{(-1)^k x^{2k+1}}{(2k+1)!}$ $(k \geq 0)$

i) $x_k = \frac{x^{2k+1}}{(2k+1)!}$ $(k \geq 0)$

j) $x_k = \frac{(-1)^k x^{2k+1}}{(2k+1)!}$ $(k \geq 0)$

3.41. Скласти програми для обчислення сум:

a) $S_n = 1 + 2 + 3 + \ldots + n$;

b) $S_n = 1 - 2 + 3 + \ldots + (-1)^n n$;

c) $S_n = 1 - \frac{1}{2} + \frac{1}{3} - \cdots + (-1)^{n-1}\frac{1}{n}$

d) $S_n = \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{(n-1)\cdot n}$

e) $S_n = 1 - \frac{2}{1!} + \frac{4}{2!} + \cdots + \frac{(-2)^n}{n!}$;

f) $S_n = \frac{1}{2} - \frac{2}{3} + \frac{3}{4} - \cdots + \frac{(-1)^n(n-1)}{n}$;

g) $S_n = \frac{1}{1!!} + \frac{2}{3!!} + \frac{4}{5!!} + \cdots + \frac{2^n}{(2n+1)!!}$;

3.42. Створити програми для обчислення добутків:

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

i) $P_n = \prod_{i=2}^{n}\left(1-\frac{1}{i^2}\right) = \left(1-\frac{1}{2^2}\right)\left(1-\frac{1}{3^2}\right)\ldots\left(1-\frac{1}{n^2}\right)$;

j) $P_n = \prod_{i=1}^{n}\left(2+\frac{1}{i!}\right)$;

k) $P_n = \prod_{i=1}^{n}\frac{i+1}{i+2}$;

l) $P_n = \prod_{i=1}^{n} \frac{1}{i+1!}$;

m) $P_n = \prod_{i=1}^{n} \frac{1}{1+i^i}$;

n) $P_n = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \ldots$, n – співмножників.

3.43. Створити програми для обчислення ланцюгових дробів:

a) $1 + \frac{1}{1+\frac{1}{1+\ddots+\frac{1}{1}}}$;

b) $1 + \frac{1}{2+\frac{1}{2+\ddots+\frac{1}{2}}}$;

c) $\mathrm{n} + \frac{1}{(\mathrm{n}-1)+\frac{1}{(\mathrm{n}-2)+\ddots+\frac{1}{2}}}$;

d) $\lambda_{\mathrm{n}} = 2 + \frac{1}{6+\frac{1}{10+\ddots+\frac{1}{4n+2}}}$..

3.44. Для перших n чисел послідовності чисел Фібоначчі F_n переконатися у справедливості твердження: F_n може бути простим тільки для простих n (за виключенням $n = 4$).

3.45. Числами трибоначчі називається числова послідовність $\{T_k : k \geq 0\}$, задана рекурентним співвідношенням третього порядку:

$$T_0 = 0, T_1 = T_2 = 1, T_k = T_{k-1} + T_{k-2} + T_{k-3}\,, \qquad k \geq 3.$$

Скласти програму для обчислення T_n.

3.46. Послідовністю Падована називається числова послідовність $\{P_k : k \geq 0\}$, задана рекурентним співвідношенням третього порядку:

$$P_0 = P_1 = P_2 = 1, P_k = P_{k-1} + P_{k-3}\,, \qquad k \geq 3.$$

Скласти програму для обчислення P_n. Для всіх $n \leq N$ безпосередньою перевіркою переконатися, що послідовність Падована задовольняє таким рекурентним формулам:

a) $P_n = P_{n-1} + P_{n-5}$;

b) $P_n = P_{n-2} + P_{n-4} + P_{n-8}$;

c) $P_n = 2P_{n-2} - P_{n-7}$;

d) $P_n = 4P_{n-5} + P_{n-14}$.

3.47. Скласти програму для обчислення суми перших n членів:

a) послідовності Фібоначчі;

b) послідовності трибоначчі;

c) послідовності Падована.

3.48. Знайти k-ту цифру послідовності:

a) 110100100010000 ... , у якій виписані підряд степені 10;

b) 123456789101112 ... , у якій виписані підряд усі натуральні числа;

c) 149162536 ... , у якій виписані підряд квадрати всіх натуральних чисел;

d) 01123581321 ... , у якій виписані підряд усі числа Фібоначчі.

3.49. Скласти програми для обчислення визначників порядку n:

a) $\begin{vmatrix} 2 & 3 & 0 & \ldots & 0 \\ 1 & 2 & 3 & \ldots & 0 \\ 0 & 1 & 2 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 2 \end{vmatrix}$;

b) $\begin{vmatrix} 2 & 1 & 0 & \ldots & 0 \\ 1 & 2 & 1 & \ldots & 0 \\ 0 & 1 & 2 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 2 \end{vmatrix}$;

c) $\begin{vmatrix} 3 & 2 & 0 & \dots & 0 \\ 1 & 3 & 2 & \dots & 0 \\ 0 & 1 & 3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 3 \end{vmatrix}$;

d) $\begin{vmatrix} 7 & 5 & 0 & \dots & 0 \\ 2 & 7 & 5 & \dots & 0 \\ 0 & 2 & 7 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 7 \end{vmatrix}$;

e) $\begin{vmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ -1 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & -1 & 0 \end{vmatrix}$;

f) $\begin{vmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix}$;

g) $\begin{vmatrix} a+b & ab & 0 & 0 & \dots & 0 \\ 1 & a+b & ab & 0 & \dots & 0 \\ 0 & 1 & a+b & ab & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & a+b \end{vmatrix}$;

h) $\begin{vmatrix} 1 & 2 & 0 & 0 & 0 & \dots & 0 & 0 \\ 3 & 4 & 3 & 0 & 0 & \dots & 0 & 0 \\ 0 & 2 & 5 & 3 & 0 & \dots & 0 & 0 \\ 0 & 0 & 2 & 5 & 3 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 & \dots & 5 & 3 \\ 0 & 0 & 0 & 0 & 0 & \dots & 2 & 5 \end{vmatrix}$;

i) $\begin{vmatrix} a & 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & a & 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & a & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 & a \end{vmatrix}$;

j) $\begin{vmatrix} 1+x^2 & x & 0 & 0 & \dots & 0 & 0 \\ x & 1+x^2 & x & 0 & \dots & 0 & 0 \\ 0 & x & 1+x^2 & x & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & x & 1+x^2 \end{vmatrix}$.

3.50. Створити програми для обчислення сум:

a) $\sum_{k=1}^{n} 2^k a_k$, де $a_1 = 0,\ a_2 = 1,\quad a_k = a_{k-1} + k a_{k-2},\quad k \geq 3$;

b) $S_n = \sum_{k=1}^{n} \frac{3^k}{a_k}$, де $a_1 = 1,\ a_2 = 1,\quad a_k = \frac{a_{k-1}}{k} + a_{k-2},\quad k \geq 3$;

c) $S_n = \sum_{k=1}^{n} \frac{k!}{a_k}$, де $a_1 = 1,\ a_2 = 1,\quad a_k = a_{k-1} + \frac{a_{k-2}}{2^k},\quad k \geq 3$;

d) $S_n = \sum_{k=1}^{n} k!\, a_k$, де $a_1 = 0,\ a_2 = 1,\quad a_k = a_{k-1} + \frac{a_{k-2}}{(k-1)!},\quad k \geq 3$;

e) $S_n = \sum_{k=1}^{n} \frac{a_k}{2^k}$, де $a_1 = a_2 = a_3 = 1,\quad a_k = a_{k-1} + a_{k-3}, k \geq 4$;

f) $\sum_{k=1}^{n} 2^k\, a_k$, де $a_1 = 1,\quad a_k = k a_{k-1} + \frac{1}{k},\quad k \geq 2$.

3.51. Створити програми для обчислення сум:

a) $S_n = \sum_{k=1}^{n} \frac{2^k}{a_k + b_k}$,

де $\begin{cases} a_1 = 0, a_2 = 1, \\ a_k = \frac{a_{k-1}}{k} + a_{k-2} b_k \end{cases} \quad \begin{cases} b_1 = 1, b_2 = 1, \\ b_k = b_{k-1} + a_{k-1} \end{cases} \quad k \geq 3$;

b) $S_n = \sum_{k=1}^{n} \frac{a_k b_k}{(k+1)!}$,

де $\begin{cases} a_1 = u, \\ a_k = 2b_{k-1} + a_{k-1} \end{cases} \quad \begin{cases} b_1 = v, \\ b_k = 2a_{k-1}^2 + b_{k-1} \end{cases} \quad k \geq 2$;

u, v – задані дійсні числа;

c) $\sum_{k=1}^{n} \frac{2^k}{\left(1 + a_k^2 + b_k^2\right) k!}$,

де $\begin{cases} a_1 = 1, \\ a_k = 3b_{k-1} + 2a_{k-1} \end{cases} \quad \begin{cases} b_1 = 1, \\ b_k = 2a_{k-1} + b_{k-1} \end{cases} d, \quad k \geq 2$;

d) $\sum_{k=1}^{n} \left(\frac{a_k}{b_k}\right)^k$

де $\begin{cases} a_0 = 1, a_1 = 2, \\ a_k = a_{k-2} + \frac{b_k}{2}, \end{cases} \quad \begin{cases} b_0 = 5, b_1 = 5, \\ b^k = b_{k-2}^2 - a_{k-1} \end{cases} \quad k \geq 2$;

e) $\sum_{k=1}^{n} \frac{a_k}{1 + b_k}$,

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

3.52. Створити програми для обчислення добутків:

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

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

b) $P_n = \prod_{k=0}^{n} a_k b_k$,

де $\begin{cases} a_1 = 1, \\ a_k = \left(\sqrt{b_{k-1}} + a_{k-1}\right)/5' \end{cases}$

$\begin{cases} b_1 = 1, \\ b_k = 2b_{k-1} + 5a_{k-1}^2, \end{cases} \quad k \geq 2.$

3.53. Для довільного цілого числа $m > 1$ знайти найбільше ціле k, при якому $4^k < m$.

3.54. Для заданого натурального числа n одержати найменше число вигляду 2^r, яке перевищує n.

3.55. Дана непорожня послідовність із натуральних чисел, за якою йде 0. Обчислити суму тих із них, порядкові номери яких – числа Фібоначчі.

3.56. Маємо дійсне число a. Скласти програму для обчислення:

a) серед чисел $1{,}1+\frac{1}{2}, 1+\frac{1}{2}+\frac{1}{3}$, … першого, більшого за a;

b) такого найменшого n, що $1+\frac{1}{2}+\ldots+\frac{1}{n}>a$.

3.57. Скласти програми обчислення:

a) найбільшого числа Фібоначчі, яке не перевищує число a;

b) номера найменшого числа Фібоначчі, яке більше від числа a;

c) суми всіх чисел Фібоначчі, які не перевищують число a.

3.58. Послідовність задана рекурентним співвідношенням

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

Створити програму для знаходження найбільшого члена цієї послідовності та його номера, який не перевищує число a.

3.59. За допомогою розкладу функції в ряд Тейлора обчислити з точністю $\varepsilon > 0$ її значення для заданого значення x:

a) $y = \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$;

b) $y = \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$;

c) $y = shx = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots$;

d) $y = chx = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots$;

e) $y = \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$; $(|x| < 1)$;

f) $y = \frac{1}{1+x} = 1 - x + x^2 - x^3 + \ldots$; $(|x| < 1)$;

g) $y = \ln\frac{1+x}{1-x} = 2 \cdot \left[\frac{x}{1} + \frac{x^3}{3} + \frac{x^5}{5} + \ldots\right]$; $(|x| < 1)$;

h) $y = \frac{1}{(1+x)^2} = 1 - 2 \cdot x + 3 \cdot x^2 - \ldots$; $(|x| < 1)$;

i) $y = \frac{1}{(1+x)^3} = 1 - \frac{2 \cdot 3}{2} \cdot x + \frac{3 \cdot 4}{2} \cdot x^2 - \frac{4 \cdot 5}{2} \cdot x^3 + \ldots$; $(|x| < 1)$;

j) $y = \frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + \ldots$; $(|x| < 1)$;

k) $y = \sqrt{1+x} = 1 + \frac{1}{2} \cdot x - \frac{1}{2 \cdot 4} \cdot x^2 + \frac{1 \cdot 3}{2 \cdot 4 \cdot 6} \cdot x^3 - \ldots$, $(|x| < 1)$;

l) $y = \frac{1}{\sqrt{1+x}} = 1 - \frac{1}{2} \cdot x + \frac{1 \cdot 3}{2 \cdot 4} \cdot x^2 - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \cdot x^3 + \ldots$; $(|x| < 1)$;

m) $y = \arcsin x = x + \frac{1}{2} \cdot \frac{x^3}{3} + \frac{1 \cdot 3}{2 \cdot 4} \cdot \frac{x^5}{5} + \ldots$. $(|x| < 1)$.

Порівняти отримані результати із значеннями відповідних функцій з бібліотеки `math`

3.60. Дано дійсні числа $x, \varepsilon, (x \neq 0, \varepsilon > 0)$. Обчислити з точністю ε нескінченну суму і вказати кількість врахованих доданків:

a) $\sum_{k=0}^{\infty} \frac{x^{2k}}{2k!}$;

b) $\sum_{k=0}^{\infty} \frac{(-1)^k x^k}{(k+1)^2}$;

c) $\sum_{k=0}^{\infty} \frac{x^{2k}}{2^k k!}$;

d) $\sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{k!(2k+1)}$.

3.61. Для заданого значення x $(|x| < (\sqrt{5}-1)/2)$ з точністю $x, \varepsilon, (x \neq 0, \varepsilon >$

0) обчислити генератрису послідовності чисел Фібоначчі:

$$\frac{x}{1-x-x^2} = \sum_{n=0}^{\infty} F_n\, x^n,$$

де $\{F_n, n \geq 0\}$ – послідовність чисел Фібоначчі.

3.62. Для заданих $\varepsilon > 0$ і $x > 0$ скласти програму наближеного обчислення кореня k-го степеня $y(x) = \sqrt[k]{x}$ з числа x, використовуючи відповідні рекурентні співвідношення і границю $\lim\limits_{n\to\infty} x_n = \sqrt[k]{x}$.

a) корінь квадратний $y(x) = \sqrt{x}$, $k = 2$:

$$\begin{cases} x_0 = \dfrac{x}{2}, \\ x_n = \dfrac{1}{2}\left(x_{n-1} + \dfrac{x}{x_{n-1}}\right), n \geq 1; \end{cases}$$

b) корінь третього степеня $y(x) = \sqrt[3]{x}, k = 3$:

$$\begin{cases} x_0 = \dfrac{x}{3}, \\ x_n = \dfrac{1}{3}\left(2x_{n-1} + \dfrac{x}{x_{n-1}^2}\right), n \geq 1; \end{cases}$$

c) кореня четвертого степеня $y(x) = \sqrt[4]{x}, k = 4$:

$$\begin{cases} x_0 = \dfrac{x}{4}, \\ x_n = \dfrac{1}{4}\left(3x_{n-1} + \dfrac{x}{x_{n-1}^2}\right), n \geq 1; \end{cases}$$

d) кореня п'ятого степеня $y(x) = \sqrt[5]{x}, k = 5$:

$$\begin{cases} x_0 = \dfrac{x}{5}, \\ x_n = \dfrac{1}{5}\left(4x_{n-1} + \dfrac{x}{x_{n-1}^4}\right), \qquad n \geq 1. \end{cases}$$

Під наближеним з точністю $\varepsilon > 0$ значенням $y(x) = \sqrt[k]{x}$ вважати значення такого члена x_N послідовності, що

a) $|x_N - x_{N-1}| < \varepsilon$; b) $|x_N^k - x| < \varepsilon$.

Порівняти отримані результати із значеннями відповідних функцій з бібліотеки `math`.

3.63. Скласти програму наближеного обчислення з заданою точністю границі послідовності, заданої рекурентним співвідношенням

a) $a_0 = 0, a_i = \frac{a_{i-1}+1}{a_{i-1}+2}, i = 1,2\ldots;$ b) $a_0 = 1,\ a_i = \frac{2-a_{i-1}^3}{5},\quad i \geq 1.$

3.64. Задане дійсне число x. Послідовність $a_1, a_2, \ldots$ утворена за таким законом $a_1 = x$: далі для $i \geq 2$ виконано:

a) $a_i = \sqrt{|4 \cdot a_{i-1}^2 - 2x|}$;
b) $a_i = \frac{16+x}{1+|a_{i-1}^3|}$;
c) $a_i = 2 \cdot a_{i-1} + \frac{x}{4+a_{i-1}^2}$;
d) $a_i = 3 + \frac{1}{2^i}\cos^2(a_{i-1} - x)$.

Скласти програму для обчислення границі послідовності a_n із заданою точністю.

3.65. Задані рекурентні співвідношення:

$$x_1 = a, y_1 = b,$$
$$x_i = \frac{1}{2}(x_{i-1} + y_{i-1}), \quad y_i = \sqrt{x_{i-1} \cdot y_{i-1}}, \quad i \geq 2 \quad (a > b > 0).$$

Скласти програму для обчислення першого члена послідовності x_i такого, що $|x_i - y_i| < \varepsilon$, $(\varepsilon > 0)$.

3.66. Скласти програму наближеного обчислення числа π:

a) за формулою Грегорі

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

b) використовуючи добуток

$$\frac{2}{\pi} = \sqrt{\frac{1}{2}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{\frac{1}{2}}} \cdot \sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{\frac{1}{2}}}}$$

c) за формулою Валліса

$$\frac{\pi}{2} = \frac{2 \cdot 2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdot 8}{1 \cdot 3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdot 9} \cdot \ldots$$

Проаналізувати, який з методів дає кращий результат за меншу кількість ітерацій.

3.67. Методом ділення відрізка навпіл, знайти корінь рівняння $\sin x = x/3$ із заданою точністю ε на відрізку $[1.6, 3]$.

3.68. Із заданою точністю ε знайти корінь рівняння $x^3 + 4x^2 + x - 6$, що належить відрізку $[0,2]$.

§4 СПИСКИ ТА КОРТЕЖІ

Списки

Список у Python – це впорядкована колекція (тобто індексована послідовність) об'єктів довільних типів.

Елементи списку у літералах перераховуються через кому і записуються у квадратних дужках

```
[0, 1, 2, 3, 4, 5]
['first', 'second', 100, 1.234]
```

Списки у Python належать до **змінюваних** (**mutable**) типів.

Створення списків

Існує кілька способів створення списків. Розглянемо їх

1. Створення списку за допомогою літерала. Власне потрібно просто перерахувати елементи списку у **квадратних дужках через кому**

```
empty_list = []              # Порожній список

l = ['1', '2', 3.0, 4]       # Список з 4-х
                             # різнотипових елементів

M = [[1, 2, 3], [4, 5, 6]] # Список, що складається
                             # з двох списків
```

2. За допомогою перетворення у список іншої колекції використовуючи ключове слово `list`.

```
lst = list(collection)
```

тут `lst` – новий список, що будується на базі колекції `collection`.

```
l = list(range(5)) # Список [0, 1, 2, 3, 4]
s = list('list')   # Список ['l', 'i', 's', 't']
```

3. За допомогою оператора створення списку (також будемо використовувати термін спискоутворення). **Оператор створення списку** – це спосіб побудови нового списку на базі іншої колекції, до всіх елементів якої застосовується деякий вираз. Його синтаксис такий

```
new_lst = [expr(i) for i in collection if condition]
```

тут `new_lst` – новий список, що створюється, `collection` – деяка колекція.

Оператор створення списку працює таким чином:
змінна `i` послідовно пробігає всі елементи колекції `collection`. Якщо для поточного значення `i` виконується умова `condition`, то в список додається елемент `expr(i)`, що залежить від поточного значення ітератора.

Наприклад, для того, щоб створити список, що складається з квадратів натуральних чисел, що не перевищують 30 і є кратними 5: [25, 100, ...] потрібно виконати таку інструкцію

```
>>> l = [i ** 2 for i in range(5, 31) if i % 5 == 0]
>>> print(l)
[25, 100, 225, 400, 625, 900]
```

Якщо в операторі створення списку умова відсутня, то блок `if` опускають

Для прикладу створимо список, що складається з квадратів натуральних чисел, що не перевищують 4

```
>>> l = [i ** 2 for i in range(5)]
>>> print(l)
[0, 1, 4, 9, 16]
```

Індексація

Список, це послідовність, у якій всі елементи занумеровані, **починаючи з 0**. Номер елементу списку називається **індексом.** Наприклад, у такому списку

```
l = [1, 2, 3, 4]
```

`1` має індекс 0, `2` – індекс 1, `3` – індекс 2, `4` – індекс 3. Звернемо увагу на те, що останній елемент має номер на одиницю менший ніж кількість елементів у списку.

У Python існує механізм по-елементної роботи зі списком.

Для того, щоб звернутися до відповідного елемента списку, необхідно вказати номер цього елементу у квадратних дужках.

```
>>> l[0]
```

```
1
>>> l[3]
4
>>> l[4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Звернення до елементу списку за індексом надає прямий доступ до цього елементу у пам'яті. Таким чином, цей елемент можна замінити на інший елемент. При цьому список зміниться, оскільки списки належать до змінюваних типів даних

```
>>> l = [1, 2, 3, 4]
>>> l[0] = 111
>>> l
[111, 2, 3, 4]
```

Елемент списку може брати участь у різних операціях згідно з тими правилами які дозволені для об'єктів його типу

```
>>> l = [1, 2, 3, 4]
>>> suma = l[0] + l[1] + l[2] + l[3]
>>> print(suma)
10
```

Звернення до елементу списку може відбуватися з використанням **від'ємних індексів**. При цьому нумерація рахується з кінця списку з урахування, що –1-й елемент це останній елемент списку, –2-й – передостанній і т.д.

```
>>> l = [1, 2, 3, 4]
>>> l[-3]
2
>>> l[-1] = 555
>>> l
[1, 2, 3, 555]
```

Зрізи

Крім індексів, для прямого доступу до елементів списків у Python існують **зрізи**. Фактично зріз списку – це фрагмент вихідного списку.

Нехай `lst` деякий список. Тоді зріз

```
lst[start : end : step]
```

це підпослідовність елементів списку `lst`, що починається з елементу з індексом `start`, закінчується елементом з індексом `end-1` та будується з кроком `step`.

```
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[2:7:2]
[3, 5, 7]
```

Для аргументів зрізу можуть використовуватися типові значення (eng. by default). Так для параметра `start` типовим значенням є 0, для `end` – кількість елементів у списку, для `step` – значення 1. Отже, якщо необхідно побудувати зріз, що має для одного (або кількох) з аргументів типове значення, його можна опускати в коді. При чому параметр `step` можна опускати разом з другою двокрапкою. Наприклад

```
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l[:3]
[1, 2, 3]
>>> l[::2]
[1, 3, 5, 7, 9, 11, 13]
```

Якщо при побудові зрізу, значення `start >= end` або діапазон значень індексів виявиться за межами списку, то зріз буде порожнім

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[5:3]
[]
>>> l[10:20]
[]
>>> l[1:1]
[]
```

Параметри зрізу також можуть бути від'ємними. У цьому випадку нумерація для `start` та `end` рахується з кінця починаючи від `-1`. Якщо ж `step` є від'ємним, то значення `start` та `end` фактично міняються місцями

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[::-1]
[6, 5, 4, 3, 2, 1]
>>> l[-1:1:-1]
[6, 5, 4, 3]
```

Використовуючи зрізи можна не лише видобувати підсписок, але і модифікувати список, додаючи або видаляючи елементи

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[0:0] = [0]    # вставляється список [0]
                    # перед нульовою позицією
>>> l
[0, 1, 2, 3, 4, 5, 6]
>>> l[3:] = []    # видаляються всі елементи починаючи з 3-го
>>> l
[0, 1, 2]
>>> l[3:] = [88, 99] # додаються в кінець список два елементи
>>> l
[0, 1, 2, 88, 99]
```

Операції над списками

Таблиця 4.1. Конкатенація і зчеплення

Операція	Опис
`l1 + l2`	конкатенація `l1` та `l2`, тобто створення нового списку, що складається з першого списку до кінця якого дописано другий список.
`l * n` `n * l`	`n` зчеплених копій списка `l`

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> s = [10, 20, 30, 40, 50, 60]
>>> l + s
[1, 2, 3, 4, 5, 6, 10, 20, 30, 40, 50, 60]
>>> l * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Таблиця 4.2. Перевірка входження елементу у список

Операція	Опис
`x in l`	Повертає значення **`True`**, якщо елемент `x` входить до списку `l`
`x not in l`	Повертає значення **`True`**, якщо елемент `x` **не входить** до

	списку `l`

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> 2 in l
True
>>> 222 in l
False
>>> 222 not in l
True
```

Таблиця 4.3. Аналіз списку

Операція	Опис
`len(l)`	Довжина `l`, тобто кількість елементів у списку.
`min(l)`	Найменший елемент списку `l`
`max(l)`	Найбільший елемент списку `l`
`l.index(x,i,j)`	Індекс першого входження `x` до `l`. (починаючи з індекса `i` та перед індексом `j`) Параметри `i` та `j` мають типові значення 0 та кількість елементів у списку відповідно.
`l.count(x)`	Кількість входжень `x` до `s`

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> len(l)
6
>>> min(l)
1
>>> l.index(3)
2
>>> l = [1, 2, 2, 2, 2, 3]
>>> l.count(2)
4
```

Таблиця 4.4. Модифікація списку

Операція	Опис
`s.append(x)`	Додає `x` у кінець списку `l`.
`l.clear()`	Вилучає всі елементи зі списку `l`
`l.copy()`	Повертає копію списку `l`
`l.extend(t)`	Розширює список `l` списком `t` (додає в кінець списку `l` список `t`)
`l.insert(i,x)`	Вставляє `x` у `l` у позицію, задану `i`
`l.pop(i)`	Повертає елемент з позиції `i` та вилучає його зі

	списку `l`
`l.pop()`	Повертає останній елемент списку `l`, та вилучає його зі списку `l`
`l.remove(x)`	Видаляє перший такий елемент `l`, для якого `l[i] == x`
`l.reverse()`	Переставляє елементи `l` у зворотному порядку
`l.sort()`	Впорядковує список `l` за неспаданням

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l.extend([20, 29])
>>> l
[1, 2, 3, 4, 5, 6, 20, 29]
>>> l.pop()
29
>>> l
[1, 2, 3, 4, 5, 6, 20]
>>> l.clear()
>>> l
[]
```

Наведені у таблиці 4.4 операції для списків можна реалізувати використовуючи зрізи та/або інші операції. Наприклад, інструкцію додавання елементу до списку

```
l.append(x)
```

можна записати як

```
l[len(l):len(l)] = [x]
```

Інструкцію

```
l.extend(t)
```

можна записати як

```
l[len(l):len(l)] = t
```

Інструкцію видалення всіх елементів зі списку

```
l.clear()
```

можна записати як

```
l[::] = []
```

Аналогічно можна записати й інші інструкції, що рекомендується виконати читачу самостійно.

Особливості роботи зі списками

Як було сказано раніше, списки відносяться до змінюваних (**mutable**) типів даних. Отже, через змінну, що є ім'ям списку надається безпосередній доступ, до пам'яті де список зберігає свої дані.

Інколи, особливо у новачків, не розуміння цього призводить до неочікуваних наслідків.

Розглянемо для прикладу такий код

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> s = l
>>> s.pop()
6
>>> l
[1, 2, 3, 4, 5]
```

Як бачимо список `l` змінився – зник елемент 6. Виникає цілком логічне запитання – Чому? Адже ми нічого не робили зі списком `l`.

Пояснення цього «феномену» полягає розумінні того як влаштовані змінні у Python. А саме: як було зазначено у першому параграфі – змінні це адреси пам'яті (посилання). Інструкція `s = l` провела копіювання адрес, а не списків і тому фактично `s` і `l` це два різних імені одного списку. Змінивши список `s` ми змінили і список `l`.

Якщо наведена вище поведінка не є бажаною, то список треба копіювати або інструкцією `copy` або з допомогою зрізів

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> s = l.copy()     # s = l[:]
>>> s.pop()
6
>>> l
[1, 2, 3, 4, 5, 6]
>>> s
[1, 2, 3, 4, 5]
```

Як бачимо у цьому випадку список лишився незмінним.

Введення списків з клавіатури

Список можна виводити на екран за допомогою інструкції `print`. Проте в Python не передбачено готової інструкції для введення списків з клавіатури.

Для цього використовується по-елементне введення, наприклад використовуючи цикл по проміжку значень.

```
# Блок введення списку з клавіатури
N = int(input("Кількість елементів списку "))
l = []              # Створюємо порожній список
for i in range(N):
    x = float(input("Задайте %d-й елемент списку " % (i)))
    l.append(x)    # Додаємо елемент x у список
# ==================================
# Виведення списку на екран
print(l)
```

Обхід списків

1. Ітерування списку

Оскільки список є колекцією, то всі його елементи можна послідовно перебрати використовуючи цикл **for**. Нехай маємо список `lst`. Тоді цикл

```
for elem in lst:
    process_iteration
```

послідовно виконує `process_iteration` для всіх елементів `elem` списку. Наприклад, знайдемо суму елементів списку, що містить натуральні числа від 1 до 5.

```
l = [1, 2, 3, 4, 5]
suma = 0                # Змінна у якій підраховується сума
for elem in l:
    suma += elem        # Додаємо кожний елемент списку
print(suma)
```

Приклад 4.1. Знайти найбільший елемент списку.

Розв'язок. Хоча для розв'язання цієї задачі у Python є вбудована функція `max`, розв'яжемо цю задачу без її використання, для того, щоб краще зрозуміти принципи роботи зі списками. Схожу задачу, проте без використання списків ми вже раніше розв'язували, тому наведемо програму без додаткових пояснень самого алгоритму

```
# Блок введення списку з клавіатури
N = int(input("Кількість елементів списку "))
l = []              # Створюємо порожній список
```

```
for i in range(N):
    x = float(input("Задайте %d-й елемент списку " % (i)))
    l.append(x)    # Додаємо елемент x у список
# ==================================
# Знаходження максимального елементу
max = l[0]  # 0-й елемент списку покладемо найбільшим
for i in range(1, len(l)):  # Проходимо всі елементи списку
    if max < l[i]:  # Якщо поточний максимум
                    # менший за поточний елемент
        max = l[i]  # змінюємо поточний максимум
print("Найбільних елемент списку - ", max)
```

2. Обхід списку за індексами

Крім по-елементного проходу списку, елементи списку можна перебрати за їхніми індексами. Нехай `lst` деякий список. Тоді цикл

```
for i in range(len(lst)):
    process_iteration      # Тут використовують lst[i]
```

послідовно виконує `process_iteration` для всіх елементів списку, звертання до яких потрібно за їхніми індексами, тобто `lst[i]`.

Приклад 4.2. Знайти індекс найбільшого елементу списку.

Розв'язок. Оскільки алгоритм введення списку з клавіатури дослівно повторює такий з попередньої задачі, то просто зазначимо його схематично – при необхідності замість крапок потрібно вставити фрагмент коду з попереднього прикладу. У циклі будемо проходити всі елементи по індексах. Будемо запам'ятовувати не лише найбільший поточний елемент, але і його індекс

```
# Блок введення списку з клавіатури
.....
# Знаходження максимального елементу разом з його індексом
max_elem = l[0] # 0-й елемент списку покладемо найбільшим
max_index = 0   # Індекс найбільшого елементу списку
for i in range(1, len(l)): # Проходимо всі елементи списку
    if max_elem < l[i]: # Якщо поточний максимум
                        # менший за поточний елемент
        max_elem = l[i] # змінюємо поточний максимум
        max_index = i   # змінюємо індекс максимального ел-ту
print("Найбільних елемент %d має індекс %d" % (max_elem, max_index))
```

3. Обхід списку за допомогою функції `enumerate`.

Функція `enumerate`, дозволяє пробігати циклом `for` список як колекцію і при цьому надає доступ до індексів елементів списку

```
for i, elem in enumerate(lst):
    process_iteration # Тут використовують i та elem
```

Тут `i` – індекс елементу списку, `elem` – поточний елемент колекції.

Отже, цикл пошуку найбільшого елемента у попередньому прикладі можемо переписати таким чином

```
for i, elem in enumerate(l):
    if max_elem < elem:
        max_elem = elem
        max_index = i
```

Приклад 4.3. Знайти суму елементів дійсного вектора.

Розв'язок. Як відомо вектор – це впорядкований набір дійсних чисел. Отже цю задачу зручно розв'язати, використовуючи список. Як і у попередньому прикладі, розіб'ємо задачу на два кроки – введення вектора з клавіатури і визначення суми елементів заданого вектора. Аналогічно попередньому прикладу код введення списку з клавіатури зазначимо схематично. Суму елементів вектора знайдемо пробігаючи його елементи циклом по колекції.

Таким чином, програма буде мати вигляд

```
# Блок введення списку з клавіатури
.....
# Обчислення суми елементів заданого вектора
suma = 0          # змінна у якій підраховується сума
for elem in l:    # пробігаємо список по-елементно
    suma += elem # додаємо до суми значення компоненти
print("Сума елементів вектора =", suma)
```

Моделювання роботи з матрицями за допомогою списків

Як відомо, будь-яку матрицю можна трактувати як вектор, що складається з векторів.

$$\begin{pmatrix} a_{00} & \dots & a_{0,N-1} \\ \dots & \dots & \dots \\ a_{N-1,0} & \dots & a_{N-1,N-1} \end{pmatrix} = \begin{pmatrix} (a_{00} & \dots & a_{0,N-1}) \\ \dots & \dots & \dots \\ (a_{N-1,0} & \dots & a_{N-1,N-1}) \end{pmatrix} =$$

Тоді, очевидно, що з матрицями можна працювати у Python як зі списками, елементами яких є списки дійсних чисел. Наприклад, матрицю

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

можна зобразити списком з двох елементів, кожен з яких містить відповідний рядок матриці

```
M = [ [1, 2, 3], [4, 5, 6] ]
```

Таке зображення списків наближає роботу з ними до роботи зі звичайними матрицями. Фактично єдиною відмінністю буде те, що в математиці, як правило номери рядків і стовпчиків починаються з одиниці, а елементи списків нумеруються з нуля. Отже, для того, щоб отримати елемент матриці розташований на перетині i-го рядка і j-го стовпчика необхідно брати відповідні елементи списків з номерами рядка і стовпчика $i-1$ і $j-1$ відповідно.

Наприклад, якщо для вищенаведеної матриці треба взяти елемент, що знаходиться у другому рядку і першому стовпчику (що має значення 4) необхідно взяти елемент `M[1][0]`.

```
>>> M = [ [1, 2, 3], [4, 5, 6] ]
>>> M[1][0]
4
```

Зображення матриць як списків, що містять вкладені списки підказує алгоритм обробки матриць, а саме за допомогою вкладених циклів по колекції. При цьому, ітератор зовнішнього циклу біжить "по рядках матриці", а ітератор внутрішнього циклу – по елементах відповідного рядка:

```
for row in M:           # row біжить по рядках матриці M
    for elem in row:    # elem пробігає елементи рядка row
        # тут можна проводити операції з
        # поточним елементом матриці M
```

Тут деяка `M` задана матриця.

Аналогічно, потрібно використовувати вкладені цикли для зчитування матриць з клавіатури

```
# Введення матриці з клавіатури
N = int(input("Кількість рядків матриці "))
L = int(input("Кількість стовпчиків матриці "))
M = []              # Створюємо порожню матрицю
for i in range(N):
```

```
    row = []       # Створюємо порожній список - рядок матриці
    # Заповнюємо рядок матриці з клавіатури
    for j in range(L):
        elem = float(input("M[%d][%d] = " % (i, j)))
        row.append(elem)
    M.append(row) # Додаємо заповнений рядок до матриці
```

Оскільки вводити дані у матрицю за допомогою вищенаведеного коду потрібно по-елементно, то таке введення матриці не зручне з точки зору практичного використання. Користувачу зручніше вводити зразу весь рядок матриці. Наведемо без пояснень (оскільки це виходить за межі розглянутих тем) код, який дозволяє вводити матрицю по рядках, розділяючи елементи рядка, символом (або кількома символами) пропуску

```
# Блок введення списку з клавіатури
N = int(input("Кількість рядків матриці "))
M = []             # Створюємо порожню матрицю
for i in range(N):
    # Вводимо рядок матриці row з клавіатури
    row = map(float, input("%d: " % i).split())
    M.append(row)  # Додаємо рядок до матриці
```

Введену матрицю можна вивести на екран за допомогою інструкції `print`. Проте у цьому випадку виведення буде здійснено як для списку

```
>>> print(M)
[[1, 2, 3], [4, 5, 6]]
```

Наведено приклад виведення матриці по рядках

```
for row in M:
    for elem in row:
        print(elem, end=" ")
    print()
```

Результатом виконання цього коду для заданої вище матриці M буде

```
1 2 3
4 5 6
```

Приклад 4.4. Визначити найменший з елементів квадратної дійсної матриці порядку N.

Розв'язок. Розіб'ємо задачу на дві частини. Перша – введення матриці з клавіатури. Друга – це пошук мінімуму у матриці. Алгоритм введення матриці з клавіатури розглянутий вище. Тому у програмі введення матриці позначимо схематично.

Для знаходження найменшого елементу матриці будемо пробігати всі елементи матриці за допомогою двох вкладених циклів, як показано вище. Використаємо додаткову змінну `min`, яка буде містити поточний мінімум серед елементів, які вже було розглянуто.

```
# Введення матриці M з клавіатури
……
# Пошук найменшого
min = M[0][0]
for row in M:
    for elem in row:
        if elem <= min:
            min = elem

print(min)
```

Приклад 4.5. Перевірити чи є задана матриця симетричною.

Розв'язок. Умова симетричності квадратної матриці має вигляд: $a_{ij} = a_{ji}$ для всіх i, j. Тому задачу можна розглядати як задачу пошуку таких i, j, що $a_{ij} \neq a_{ji}$. Якщо таких індексів немає, то матриця симетрична. У програмі використаємо змінну `res`, яка буде містити **`True`**, якщо не існує таких i, j, що $a_{ij} \neq a_{ji}$ і **`False`**, якщо такі i, j буде знайдено.

```
# Введення матриці M з клавіатури
……
# Визначення чи є матриця симетрична
n = len(M)
res = True
for i in range(n):
    for j in range(n):
        if M[i][j] != M[j][i]:
            res = False

print(res)
```

Кортежі

Кортеж у Python – це впорядкована незмінювана (immutable) колекція об'єктів довільних типів.

Елементи кортежу перераховуються через кому і записуються у круглих дужках

```
(0, 1, 2, 3, 4, 5)
('first', 'second', 100, 1.234)
```

Фактично кортеж це незмінюваний список.

Виникає запитання, навіщо потрібні кортежі, якщо є списки? Відповідь на це запитання така:

1. Кортежі надають захист колекції від несанкціонованої зміни.
2. Кортежі мають менший розмір у порівнянні зі списками.
3. Кортежі, на відміну від списків можна використовувати у якості ключа словника (певний тип даних Python, що буде розглянуто пізніше).

Робота з кортежами подібна до роботи зі списками. Власне для кортежів можна використовувати всі операції що і для списків, які не змінюють сам кортеж.

```
>>> l = (1, 2, 3, 4, 5)
>>> print(l[4])
5
```

Якщо ж потрібно змінити кортеж, то створюють новий кортеж на базі вихідного і здійснюють переприсвоєння подібно до простих типів. Спробуємо змінити останній елемент кортежу `l`.

```
>>> l[4] = 333
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Як бачимо система виводить повідомлення про помилку. Проте, зміна кортежу, через створення нового проходить без помилок

```
>>> l = l + (555, 666) # Додаємо два елементи до кортежу
>>> print(l)
(1, 2, 3, 4, 5, 555, 666)
```

Створення кортежів

Існує кілька способів створення кортежів.

1. Створення за допомогою літерала. Власне потрібно перерахувати елементи кортежу у **круглих дужках через кому**

```
empty_tuple = ()      # Порожній кортеж

one_element = (333,) # Кортеж з одного елементу.

l = ('1', '2', 3.0, 4) # Кортеж з 4-х елементів

M = ((1, 2, 3), (4, 5, 6)) # Кортеж, що складається
                           # з двох кортежів
                           # - незмінювана матриця
```

Зверніть увагу на створення кортежу з одного елементу.

```
one_element = (333,) # Кортеж з одного елементу.
```

Кома в дужках після числа **333** знаходиться не випадково. Справа у тому, що без неї Python буде розглядати дужки виключно як інструмент зміни пріоритету арифметичної операції, а не створення кортежу. І таким чином, без коми після першого елементу змінна `one_element` буде цілим числом, а не кортежом.

2. За допомогою перетворення у кортеж іншої колекції використовуючи інструкцію `tuple`.

```
tpl = tuple(collection)
```

Тут `tpl` – новий кортеж, що створюється на базі колекції `collection`.

```
t = tuple(range(5))         # Кортеж (0, 1, 2, 3, 4)
s = tuple([1, 2, 3, 4, 5]) # Кортеж (1, 2, 3, 4, 5)
```

Пакування колекцій

Пакування кортежу – це присвоєння виду

```
t = (x1, …, xN)
```

де `t` – ім'я запакованого кортежу, `x1, …, xN` – послідовність змінних (виразів, об'єктів тощо). Отже, пакування кортежу, це створення кортежу з декількох змінних або виразів.

Розпакування кортежу – це присвоєння виду

```
x1, …, xN = t
```

де `x1, …, xN` – послідовність змінних. `t` – деякий кортеж. Для розпакування кортежу, важливо, щоб кількість змінних, що стоїть зліва від знаку рівності дорівнювала кількості елементів у кортежі.

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> tpl = (a, b, c)  # Пакування кортежу
>>> print(tpl)
(1, 2, 3)
>>> x, y, z = tpl    # Розпакування кортежу у змінні x,y,z
>>> print(x, y, z)
1 2 3
```

Для списків операції пакування та розпакування також допустимі. Їхній синтаксис відрізняється лише тим, що потрібно використовувати замість круглих дужок – квадратні.

Пакування та розпакування кортежу може здійснюватися одночасно в одному виразі. Таким чином допустимі такі присвоєння:

```
x, y = u, v
```

Така інструкція присвоєння рівносильна такому ланцюгу присвоєнь

```
t = (u, v)
x, y = t
```

Приклад 4.6. Обміняти значення двох змінних.

Розв'язок. Класичне програмування для розв'язання поставленої задачі вимагає використання допоміжної змінної. Проте операції пакування/розпакування дозволяють розв'язати цю задачу (не рахуючи введення даних та виведення результату) одним рядком вихідного коду

```
a = int(input("a = "))
b = int(input("b = "))
a, b = b, a
print("a =", a, "b = ", b)
```

Операції пакування/розпакування допомагають зробити програми для обчислення елементів послідовностей заданих рекурентними співвідношеннями старших порядків значно простішими.

Приклад 4.7. Обчислити задане число Фібоначчі.

Розв'язок. Для спрощення коду, скористаємося операцією пакування/розпакування.

```
N = int(input('введіть N: '))
F2 = 1       #F2 - нульове число Фібоначчі
F1 = 1       #F1 - перше число Фібоначчі

for n in range(2, N+1):
    F2, F1 = F1, F1 + F2  # третя змінна стає непотрібною

print ('Результат', F1)
```

Приклад 4.8. Написати програму для обчислення найбільшого спільного дільника двох натуральних чисел.

Розв'язок. Ця задача була розв'язана у прикладі 3.4. Для спрощення та скорочення коду, скористаємося операціями пакування/розпакування.

```
N = int(input("Введіть перше число "))
M = int(input("Введіть друге число "))

# Запам'ятовуємо вхідні змінні
U = N
V = M
# Модифікуємо змінні U та V так, щоб U >= V
if U < V:
    U, V = V, U

# Алгоритм Евкліда з використанням пакування/розпакування
while V > 0:
    V, U = U % V, V

# Виводимо результат на екран
print("НСД(%d, %d) = %d" % (N, M, U))
```

Генератор-вирази для послідовностей

Генератор-вирази у Python дозволяють побудувати та опрацювати деяку послідовність на базі вже створеної колекції або згідно з деяким законом. Синтаксис генератор-виразу такий

```
generator = (expr(i) for i in collection if condition)
```

де `generator` – новий генератор-вираз, `collection` – деяка колекція, змінна `i` послідовно пробігає всі елементи колекції, `condition` – деяка умова, `expr(i)` – вираз, що залежить від `i`. Як і для оператора створення списку, блок з умовою не є обов'язковим.

Наступний блок коду, у якому визначено генератор-вираз `generator` виведе всі непарні числа з діапазону від 1 до 10.

```
generator = (k for k in range(1, 10, 2))
for a in generator:
    print(a)
```

Як можна помітити, генератор-вираз подібні до оператора створення списку – і той і інший будують послідовність на базі деякої колекції. Проте, на відміну від оператора створення списку, генератор-вирази не будують всю послідовність одразу – вони повертають послідовність по-елементно при кожному зверненні до об'єкту генератора. Це робить генератор-вирази незамінними у випадку коли треба обробляти великі обсяги даних якщо одночасно з цього масиву даних необхідно лише один або декілька елементів.

Приклад 4.9. Напишемо програму для наближеного обчислення числа π за формулою Грегорі

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

з використанням генератор-виразів.

Розв'язок. Знаходити число π за формулою Грегорі не найоптимальніший варіант, оскільки послідовність часткових сум для вищенаведеного ряду дуже повільно збігається. Тому для прийнятного результату треба брати значну кількість доданків. Відповідно, скористаємося генератор-виразом для побудови послідовності

$$\left\{1, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, \ldots\right\}$$

```
N = 99999  # Найбільший знаменник послід {1/k}

# Геренератор-вираз послідовності {1/k}
generator = (1.0 / k for k in range(1, N + 1, 2))

# Обчислення pi з використанням генератор-виразу
pi_4 = 0.0 # Змінна для pi/4
sign = 1.0 # Змінна для чергування знаку
```

```
for a in generator:
    pi_4 += sign * a
    sign = -sign

print(4.0 * pi_4)
```

Результатом виконання програми буде виведення такого значення

```
3.1415726535897814
```

Задачі для самостійної роботи

4.1. Дано список натуральних чисел. Знайти кількість:
a) парних компонент;
b) компонент, що діляться на 3 або 5;
c) нульових компонент;
d) компонент, що є простими числами.
e) компонент, що є числами Фібоначчі.

4.2. Дано список натуральних чисел. Використовуючи оператор створення списку, побудувати список, що містить:
a) всі парні елементи заданого списку;
b) всі елементи заданого списку, що є повними квадратами;
c) квадрати непарних компонент заданого списку.

4.3. Дано список чисел. Знайти кількість елементів списку,
a) більших b) менших
за задане число.

4.4. Обчислити суму елементів числового списку, які розташовані між позиціями максимального та мінімального елементів. Вважати, що всі елементи списку різні.

4.5. Знайти:
a) середнє арифметичне дійсного вектора;
b) довжину дійсного вектора;
c) відстань між двома точками у n-мірному евклідовому просторі;
d) скалярний добуток двох дійсних векторів.

4.6. Написати програми для:
a) множення дійсного вектора на число;
b) нормування дійсного вектора;
c) знаходження суми двох дійсних векторів;
d) обміну значень двох дійсних векторів;
e) пошуку однакових компонент.
f) перестановки компонент вектора у зворотному порядку.

4.7. Заданий список дійсних чисел $a_1, a_2, \dots, a_n$. Написати програми для знаходження:
a) $\min(a_1, a_2, \dots, a_n)$; b) $\max(|a_1|, \dots, |a_n|)$;

c) $\max(a_2, a_4, \dots)$;
d) $\min(a_1, a_3, \dots)$;
e) $\min(a_2, a_4, \dots) + \max(a_1, a_3, \dots)$;
f) $\max(a_1, a_2, a_4, a_8, \dots)$;
g) $\max(-a_1, a_2, -a_3, \dots, (-1)^n a_n)$;
h) $\max(a_1, 2a_2, \dots, na_n)$;
i) $(\min(a_1, \dots, a_n))^2 - \min(a_1^2, \dots, a_n^2)$;
j) $\max(a_1 + a_2, \dots, a_{n-1} + a_n)$.

4.8. Знайти спільну компоненту двох списків.

4.9. Перевірити чи впорядкований числовий список за зростанням.

4.10. Визначити процедуру пошуку спільної компоненти двох упорядкованих векторів.

4.11. Задані координати n точок на площині. Знайти номери двох точок, відстань між якими найбільша (вважати, що така пара точок єдина).

4.12. Дано два впорядкованих списки. Не використовуючи функцію сортування, об'єднати ці списки у один впорядкований список.

4.13. Дано список, що складається з коефіцієнтів многочлена

$$P_n(x) = a_0 * x^n + a_1 * x^{n-1} + \dots + a_{n-1} * x + a_n$$

Обчислити:

a) значення многочлена при заданому значенні x;
b) похідної від многочлена при заданому значенні x;
c) інтеграла від многочлена $P_n(x)$ на заданому відрізку.

4.14. Вважаючи, що многочлен задається списком коефіцієнтів, побудувати суму й добуток двох многочленів

$$A(x) = a_0 + a_1 x + \dots + a_n x^n \text{ і } B(x) = b_0 + b_1 x + \dots + b_m x^m.$$

4.15. Написати програми для обчислення:

a) суми всіх елементів матриці, що належать головній діагоналі;
b) суми всіх елементів, що належать побічній діагоналі;
c) суми всіх недіагональних елементів матриці;
d) кількості нульових елементів матриці.

4.16. Знайдіть норми матриці $A = \left(a_{ij}\right)_{i,j=1,\dots,n}$:

a) $\|A\| = \max\limits_{i,j=1,\dots n} |a_{ij}|$;
b) $\|A\| = \sum_{i,j=1}^{n} |a_{ij}|$;
c) $\|A\| = \max\limits_{i=1,\dots n} \sum_{j=1}^{n} |a_{ij}|$;
d) $\|A\| = \max\limits_{j=1,\dots n} \sum_{i=1}^{n} |a_{ij}|$;
e) $\|A\| = \sqrt{\sum_{i,j=1}^{n} a_{ij}^2}$;
f) $\|A\| = \sqrt[p]{\sum_{i,j=1}^{n} a_{ij}^p}$.

4.17. Напишіть програму пошуку заданого елемента в матриці.

4.18. Напишіть програму пошуку в матриці:

a) індексів усіх її ненульових елементів;
b) кількості всіх її різних елементів.

4.19. Елемент матриці називається "особливим", якщо:

1) він більший за суму інших елементів свого стовпчика;
2) у його рядку зліва від нього розташовані елементи, менші за нього, а справа – більші.

Напишіть програму пошуку кількості "особливих" елементів матриці.

4.20. Скласти програму визначення послідовності чисел Фібоначчі F_n, використовуючи матричну властивість

$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n+1} = \begin{pmatrix}F_{n+1} & F_n\\ F_n & F_{n-1}\end{pmatrix},\ n \geq 0.$$

Для правильності рівності при $n = 0$, вважати, що $F_{-1} = 0$.

4.21. Визначити програми:

a) транспонування матриці;
b) множення матриці на вектор;
c) перестановки двох заданих рядків (стовпчиків) матриці;
d) перестановки заданого рядка квадратної матриці із заданим її стовпчиком;
e) побудови цілочислової квадратної матриці порядку 7, елементами якої є числа $1, 2, ..., 49$, розташовані в ній за спіраллю;
f) видалення з матриці заданого рядка й заданого стовпчика.

4.22. Задана цілочислова квадратна матриця порядку n і цілочисельний вектор розмірності m. Замінити нулями в матриці ті елементи, які входять до заданого вектора.

4.23. Елемент матриці назвемо сідловою точкою, якщо він є найменшим у своєму рядку й водночас найбільшим у своєму стовпчику. Для заданої цілочислової матриці визначити процедуру пошуку індексів усіх сідлових точок.

4.24. Розв'язати задачу 4.1 використовуючи генератор-вирази для послідовностей.

4.25. Розв'язати задачу 3.48 використовуючи генератор-вирази для послідовностей.

4.26. Розв'язати задачу 3.56 використовуючи генератор-вирази для послідовностей.

§5 СИМВОЛИ ТА РЯДКИ

Символи та таблиці кодування

Символ – це умовний знак, що позначає певну сутність. Походить від древньо-грецького слова σύμβολον – «умовний знак» , «сигнал».

Комп'ютерний символ – це елемент множини, яка використовується для зображення та організації (комп'ютерних) даних.

До такої множини у комп'ютерних науках відносять букви різноманітних алфавітів, знаки пунктуації, цифри та різноманітні допоміжні і технічні символи.

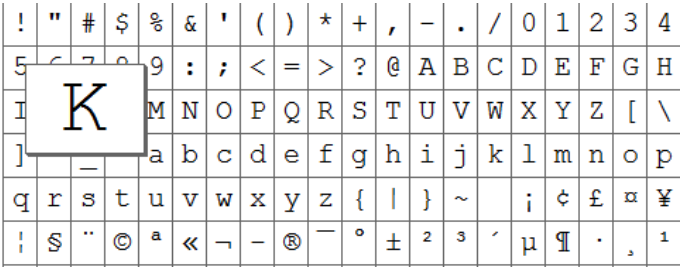


Рис. 5.1. Комп’ютерні символи

У подальшому у цьому посібнику під терміном **символ** будемо розуміти комп'ютерний символ, а вищезазначену множину символів, без обмеження загальності, будемо називати **алфавітом**.

Для збереження у пам’яті комп’ютера кожному символу співставляється деяке число, яке називають **кодом символа**, а бієктивне відображення обмеженої множини натуральних чисел у множину символів називається **таблицею кодування** символів (*eng*. **character set**).

Таблиць кодування існує багато. Нижче наведено приклад однієї з найпростіших та найпоширеніших таблиць кодування – таблиця ASCII.

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

Рис. 5.2. Фрагмент таблиці кодування ASCII

Зверніть увагу, що цифри мають коди 48..57 (а не 0,...,9), великі та маленькі літери мають різні коди, бо це різні символи.

Символи у Python

У Python символи представлені у кодуванні UTF-8.

Під терміном **символьний літерал** (або символьна стала) будемо розуміти конкретний символ із таблиці кодування. Символьні літерали беруть у апострофи або подвійні лапки, щоб відрізняти їх від ідентифікаторів, цифр та інших символьних знаків.

```
a = "a" # Змінна a набуває значення символьного літералу "a"
A = "A" # Змінна A набуває значення символьного літералу "A"
A = ";" # Змінна A набуває значення символьного літералу ";"
```

Для задавання символу апострофа у Python використовують літерал `"'"`, а для задавання символу подвійних лапок – літерал `'"'`.

Спеціальні (екрановані) символи у Python

Спеціальні або екрановані послідовності (escape-послідовності) дозволяють задавати символи, які неможливо ввести з клавіатури. Крім цього екрановані послідовності використовуються для задавання символів, що мають спеціальне призначення (наприклад, для символів `"\"`, подвійних лапок, апострофа, тощо). Екрановані послідовності починаються символом `"\"` (обернена коса риска)

Таблиця 5.1. Екрановані послідовності

Символ	Значення
\<новий рядок>	Ігнорується (продовження рядка на наступний)
\\	Обернена коса риска(зберігає \)
\'	Апостроф (зберігає ')
\"	Подвійні лапки (зберігає ")
\a	Дзвінок
\b	Крок назад
\f	Завершення форми
\n	Кінець рядка
\r	Повернення каретки
\t	Табуляція
\v	Вертикальна табуляція
\0	Символ Null

```
>>> S = "Кав\'ярня \"Львівська казка\".\nТут найкраща кава!"
>>> print(S)
Кав'ярня "Львівська казка".
Тут найкраща кава!
```

Режим екранації спеціальних символів може бути вимкнений, якщо перед рядковим літералом, що містить екрановані символи поставити символ `r`.

```
>>> S = r"Кав\'ярня \"Львівська казка\".\nТут найкраща кава!"
>>> print(S)
Кав\'ярня \"Львівська казка\".\nТут найкраща кава!
```

Як бачимо, у цьому випадку рядок містить всі символи у тому вигляді у якому його задали.

Крім стандартного способу визначення символьних літералів, описаного вище, символьні літерали можна задавати через їхні коди. Для цього використовують екрановані послідовності наведені у таблиці

Таблиця 5.2. Екрановані послідовності

Символ	Значення
\N{id}	Ідентифікатор ID бази даних Юникоду
\uhhhh	16-бітний символ Юнікоду (в 16-ковому зображенні)
\Uhhhhhhhh	32-бітний символ Юнікоду (в 16-ковому зображенні)
\xhh	16-ве значення символа (в 16-ковому зображенні)
\ooo	8-ве значення символа

```
>>> "\u0063"
'c'
```

Операції з символами

Таблиця 5.3. Найпростіші операції з символами

Операція	Значення
`ord(c)`	Код символа `c` з таблиці кодування
`chr(n)`	Символ що відповідає у таблиці кодування коду `n`
`int(c)`	Перетворення символа `c` у цифру, що йому відповідає
`str(n)`	Перетворення цифри `n` у символ

```
>>> ord("A")
65
>>> chr(65)
'A'
>>> int("5")
5
```

Символів визначені 6 стандартних відношень

```
==, !=, >, <, >=, <=
```

які фактично проводяться над кодами відповідних символів

```
>>> "A" >= "C" # код "A" менший за код "C"
False
>>> "1" < "4"  # код "1" менший за код "4"
True
```

Приклад 5.1. Вивести на екран всі великі літери латинського алфавіту.

Розв'язок. Скористаємося тим фактом, що латинські літери в таблиці кодування розташовані у алфавітному порядку. Скористаємося циклом по проміжку від коду символа "A" до коду символа "Z":

```
for i in range(ord("A"), ord("Z") + 1):
    print(chr(i), end = '')
```

Нагадаємо, що параметр `end` в інструкції `print` вказує яким символом завершувати інструкцію виведення на екран. Типовим значенням є символ `'\n'` – символ нового рядка. У цій програмі ми використали порожній символ, для того, щоб символи алфавіту були виведені в одному рядку.

Результат виконання програми

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Приклад 5.2. Визначити, чи є даний символ латинською літерою (великою або маленькою), цифрою або ні тим ні іншим.

Розв'язок. Зауважимо, що цю задачу легко розв'язати, використовуючи стандартні функції Python. Проте для глибшого розуміння роботи з символами напишемо програму без їхнього використання. Скористаємося тим фактом, що літери у таблиці кодування розташовані в алфавітному порядку, а цифри – в порядку зростання.

```
Ch = input("Задайте символ ")
if Ch >= 'a' and Ch <= 'z':
    print("%c – маленька літера" % Ch)
elif Ch >= 'A' and Ch <= 'Z':
    print("%c – велика літера" % Ch)
elif Ch >= '0' and Ch <= '9':
    print("%c – цифра" % Ch)
else:
    print("%c – ні літера, ні цифра" % Ch)
```

Приклад 5.3. Написати програму, що переводить маленькі латинські літери у відповідні великі літери (у верхній регістр).

Розв'язок. Аналогічно до попередньої задачі, розв'яжемо поточну задачу, не використовуючи готові функції Python. Аналогічно до попереднього прикладу скористаємося властивостями таблиці кодування. Помітимо, що різниця кодів між символами маленької і великої літери однакова і становить

```
ord('A') – ord('a')
```

Тоді програма має такий вигляд:

```
Ch = input("Задайте символ ")
if Ch >= 'a' and Ch <= 'z':
    UpperCh = chr(ord(Ch) + ord('A') – ord('a'))
    print("%c -> %c" % (Ch, UpperCh))
else:
    print("Введений символ не є латинською літерою")
```

Рядки

Рядок – це послідовність символів

Рядки у Python належать до незмінюваних (immutable) типів.

Аналогічно до символів, щоб відрізняти літерали рядків від ідентифікаторів, цифр та інших символьних знаків рядкові літерали (тобто послідовність символів) оточують одним з таких способів:

- апострофами або подвійними лапками для рядків, що розташовуються у одному фізичному рядку;
- потрійними апострофами або потрійними подвійними лапками для рядків, що розташовуються у кількох фізичних рядках.

```
'Рядок-літерал'
"Рядок-літерал"

'''Це рядок-літерал, який
   розташовується у кількох рядках'''

"""Це також рядок-літерал, який
   розташовується у кількох рядках"""
```

Створення рядків

Існує кілька способів створення списків. Розглянемо їх

1. Створення рядка за допомогою літерала.

```
empty_string = ""     # Порожній рядок

symbol = "a"       # Рядок, що складається з одного елементу.

few_lines = '''Рядки дуже потужний механізм
обробки текстів.'''
```

2. Створення рядка за допомогою перетворення у рядок використовуючи функцію `str`.

```
>>> s = str(12345)
>>> print(s)
'12345'
```

Робота з рядком

Фактично рядок – це кортеж символів, тому робота з рядком дуже подібна до роботи з кортежом. Отже,

- до символів рядка можна звертатися за їхніми індексами;

- всі символи рядка можна послідовно перебрати використовуючи цикл по колекції `for`;
- для рядка можна брати зріз;
- інші операції, що визначені для кортежів.

Підсумуємо у таблиці операції для рядків успадковані від кортежів

Таблиця 5.4. Операції для рядків успадковані від кортежів

Операція	Опис
`s[i]`	`i`-й символ рядка `s` Аналогічно до кортежів, `i` може бути від'ємним, тоді така операція повертає `(-i)` символ з рахуючи з кінця рядка.
`s[i:j]`	Зріз з `s` від `i` до `j` (підрядок, що починається з `i`–го символа та закінчується `j-1`-м символом)
`s[i:j:k]`	Зріз з `s` від `i` до `j` з кроком `k`
`len(s)`	довжина `s`
`min(s)`	Найменший символ рядка `s`
`max(s)`	Найбільший символ рядка `s`

```
>>> s = "Python"
>>> s[2]
't'
>>> for c in s:
        print(c)
P
y
t
h
o
n
>>> s[2:]
'thon'
>>> s[::-1]
'nohtyP'
>>> len(s)
6
>>> max(s)
'y'
```

Проте, на відміну від кортежів, для рядків визначено інструкцію `input` введення рядка з клавіатури, відомої нам з попередніх тем

```
s = input(prompt)
```

де `s` – змінна-рядок, у яку буде записано результат введення з клавіатури, `prompt` – рядок підказка, яку буде виведено на екран під час очікування введення даних з клавіатури.

Розглянемо кілька прикладів роботи з рядками.

Приклад 5.4. Перевірити, чи є рядок симетричним

Розв'язок. Рядок є симетричним якщо його запис справа на ліво збігається з записом з ліва на право. Для розв'язання цієї задачі використаємо те, що до рядка можна будувати зріз (з від'ємним кроком). Отже, отримаємо таку просту програму.

```
S = input("Введіть рядок ")

if S == S[::-1]:
    print ("Заданий рядок є симетричним")
else:
    print ("Заданий рядок НЕ є симетричним")
```

Приклад 5.5. Обчислити кількість входжень символа `a` у рядок `S`.

Розв'язок. Для розв'язання цієї задачі проітеруємо рядок по-символьно за допомогою циклу по колекції. Використаємо лічильник, який будемо збільшувати на один кожного разу коли будемо зустрічати символ `a`. Отже програма матиме вигляд

```
S = input("Введіть рядок ")
a = input("Введіть символ ")

n = 0              # Змінна лічильник
for ch in S:       # Змінна ch пробігає всі символи рядка S
    if ch == a:    # Якщо ch == a
        n += 1     # збільшуємо лічильник на одиницю

print("\"%s\" входить у рядок \"%s\" %d разів" % (a, S, n))
```

Приклад 5.6. Перевірити, чи правильно в заданому виразі розставлені круглі дужки (тобто, чи стоїть справа від кожної відкритої дужки відповідна до неї закрита дужка, а зліва від кожної закритої – відповідна до неї відкрита).

Розв'язок. Використаємо змінну `k` для підрахунку різниці кількості відкритих і закритих дужок. Додатково використаємо змінну `r`, яка буде набувати значення **`False`**, якщо закритій дужці не відповідає жодна із відкритих дужок, що їй передують. Наприклад, якщо текст має вигляд

```
"a * ( b + c ) ) * ( d"
```

то, `r = False`, хоча при цьому `k = 0`, бо кількість відкритих і закритих дужок однакова.

```
expresion = input("Введіть вираз ")
k = 0    # різниця кількості відкритих і закритих дужок
r = True # закритій дужці відповідає відкрита
for c in expresion:
    if c == '(':
        k += 1
    if c == ')':
        k -= 1
    if k < 0: # закритій дужці не відповідає відкрита
        r = False
        break

if (k == 0 and r):
    print("Дужки розставлені правильно")
else:
    print("Дужки розставлені неправильно")
```

Приклад 5.7. У заданий текст входять тільки цифри та літери. Визначити, чи правда, що сума числових значень цифр, які входять у текст, дорівнює довжині тексту.

Розв'язок. Використаємо змінну `m` для підрахунку суми цифр, що містяться у заданому тексті.

```
S = input("Введіть вираз ")
m = 0
for c in S:
    if c >= '0' and c <= '9':
        m = m + int(c) # тут використовується операція int(c)
                       # перетворення символа c у число

if len(S) == m:
    print("Так")
else:
    print("Ні")
```

Операції над рядками

Таблиця 5.5. Конкатенація і зчеплення

Операція	Опис
`s1 + s2`	конкатенація рядків `s1` та `s2`, тобто створення нового рядка, що утворений дописуванням до кінця першого рядка символів другого.
`s * n` `n * s`	`n` зчеплених копій рядка `s`

```
>>> s1 = "інформатика"
>>> s2 = " та програмування"
>>> print(s1 + s2)
інформатика та програмування
>>> print(s1 * 2)
інформатикаінформатика
```

Приклад 5.8. Надрукувати заданий текст, виключивши з нього всі знаки арифметичних операцій.

Розв'язок. Для спрощення перевірки чи є деякий символ арифметичною операцією, створимо кортеж, що містить символи арифметичних операцій. Далі, оскільки рядок це колекція, можемо пробігти його по-символьно за допомогою циклу **for**. Якщо поточний символ не є знаком арифметичної операції будемо дописувати його (за допомогою операції конкатенації) до рядка-результату.

```
expresion = input("Введіть вираз ")
operators = ('+', '-', '*', '/') # кортеж операцій
result = ""                      # рядок-результат
for c in expresion:          # пробігаємо рядок по-символьно
    if c not in operators:   # якщо поточний символ не є
                             # арифметичною операцією
        result = result + c # дописуємо його до результату

print(result)
```

Таблиця 5.6. Перевірка входження символів до рядка

Операція	Опис
`c in S`	Повертає значення **True**, якщо символ `c` міститься у рядку `S`
`c not in S`	Повертає значення **True**, якщо символ `c` не міститься у рядку `S`

```
>>> S = "інформатика"
>>> "р" in S
True
>>> "2" in S
False
>>> "2" not in S
True
```

Таблиця 5.7. Пошук за заміна

Операція	Опис
`s.count(p)`	Повертає кількість входжень рядка `p` до рядка `s`
`s.find(p,i,j)`	Індекс першого входження рядка `p` у рядок `s`. Пошук здійснюється у зрізі `s[i:j]`. Параметри `i` і `j` мають типові значення – 0 і кількість символів у рядку відповідно. Повертає значення -1 якщо `p` не знайдено.
`s.index(p,i,j)`	Індекс першого входження рядка `p` у рядок `s`. Пошук здійснюється у зрізі `s[i:j]`. Параметри `i` і `j` мають типові значення – 0 і кількість символів у рядку відповідно. Породжує помилку, якщо `p` не знайдено.
`s.replace(old, new, count)`	Повертає рядок s, у якому всі входження рядка `old` замінено рядком `new`. Параметр `count` можна опустити. Якщо задано `count`, то замінюється не більше `count` перших входжень.

```
>>> s = "Pentium 4"
>>> s.find("4")
8
>>> s.replace("4", "Pro")
'Pentium Pro'
```

Приклад 5.9. Створити програму перетворення рядка, замінивши в ньому кожну крапку трьома крапками.

Розв'язок. Для написання програми досить скористатися функцією `replace` з вищенаведеної таблиці.

```
S = input("Введіть рядок ")
S = S.replace('.', '...')
print(S)
```

Таблиця 5.8. Функції аналізу рядка

Функція	Опис
`s.isalnum()`	Повертає **`True`**, якщо всі символи рядка `s` є літерами або цифрами.
`s.isalpha()`	Повертає **`True`**, якщо всі символи рядка `s` є літерами.
`s.isdigit()`	Повертає **`True`**, якщо всі символи рядка `s` є цифрами.
`s.isidentifier()`	Повертає **`True`**, якщо рядок `s` є ідентифікатором.
`s.islower()`	Повертає **`True`**, якщо всі літери рядка `s` у нижньому регістрі
`s.isnumeric()`	Повертає **`True`**, якщо всі символи рядка `s` є числовими.
`s.isprintable()`	Повертає **`True`**, якщо всі символи рядка `s` є друкованими.
`s.isspace()`	Повертає **`True`**, якщо всі символи рядка `s` є пропусками.
`s.istitle()`	Повертає **`True`**, якщо рядок `s` є заголовком (усі слова починаються з великої літери).
`s.isupper()`	Повертає **`True`**, якщо всі літери рядка `s` у верхньому регістрі

```
>>> s = "інформатика"
>>> s.islower()
True
>>> S1 = "Pentium 4"
>>> S1.isalnum()
False
>>> S2 = "Pentium4"
>>> S2.isalnum()
True
```

Приклад 5.10. Надрукувати лише великі латинські літери, що входять до заданого рядка.

Розв'язок. Проітеруємо рядок по-символьно і скористаємося функцією `isupper()` з вищенаведеної таблиці, для того щоб вияснити чи є поточний символ великою латинською літерою.

```
S = input("Введіть рядок ")
```

```
for ch in S:
    if ch.isupper():
        print(ch)
```

Таблиця 5.9. Функції модифікації рядка

Функція	Опис
`s.capitalize()`	Повертає копію рядка `s`, у якій перший символ – велика літера, а інші – маленькі.
`s.lower()`	Повертає копію рядка `s`, у якій всі літери рядка s переведені до нижнього регістру
`s.swapcase()`	Повертає копію рядка `s`, в якій маленькі літери змінені на великі та навпаки.
`s.title()`	Повертає копію рядка `s` у форматі заголовку (усі слова починаються з великої літери).
`s.upper()`	Повертає копію рядка `s` у форматі з усіма великими літерами.

```
>>> s1 = "інформатика"
>>> s1.upper()
'ІНФОРМАТИКА'
```

Таблиця 5.10. Розбиття та склейка

Функція	Опис
`s.join(t)`	Будує та повертає рядок з усіма елементами `t` (`t` – колекція рядків – кортеж або список), між якими в якості розділювача стоїть рядок `s`
`s.split(sep)`	Повертає список, у якому рядок `s` розбито на підрядки рядками-розділювачами `sep`. Якщо `sep` не вказано, то мається на увазі рядок з довільної кількості пропусків
`s.splitlines()`	Повертає список рядків, який є розбиттям `s` на підрядки, обмежені символами кінця рядка (`\n`, `\r` або `\r\n`).

```
>>> l = ["інформатика", "та", "програмування"]
>>> separator = "____"
>>> separator.join(l)
'інформатика____та____програмування'
>>> separator1 = " "
>>> s = separator1.join(l)
>>> s
'інформатика та програмування'
>>> s.split()
['інформатика', 'та', 'програмування']
```

Приклад 5.11. Підрахувати кількість слів у введеному з клавіатури рядку.

Розв'язок. У цій задачі словом будемо називати послідовність символів, що відокремлена символами пропуску. Для розв'язання задачі достатньо застосувати до заданого рядка метод `split`, який фактично побудує список всіх слів рядка. Отже програма буде мати такий вигляд

```
S = input("Введіть рядок ")
# Наприклад, S = "  інформатика   та  програмування   "
l = S.split() # l = ['інформатика', 'та', 'програмування']
print("Кількість слів у рядку", len(l))
```

Приклад 5.12. Нехай у рядку міститься послідовність слів, що розділена одним чи декількома символами пропуску. Необхідно видалити зайві пропуски між словами, так, щоб слова розділялися лише одним символом пропуску. Також видалити всі пропуски на початку та вкінці рядка.

Розв'язок. Найпростіший спосіб розв'язання цієї задачі – розбили рядок на слова, а далі склеїти отриманий список за допомогою функції `join` застосувавши її до символа пропуска у якості розділювача.

Отже програма

```
S = input("Введіть рядок ")
                  # S = "  інформатика   та  програмування   "
l = S.split()     # l = ['інформатика', 'та', 'програмування']
result = " ".join(l)
                  # result = "інформатика та програмування"
print(result)
```

Приклад 5.13. У заданому рядку всі символи `'0'` замінити на `'1'`, а символи `'1'` на `'0'`.

Розв'язок. У цій задачі ми не можемо скористатися операцією `replace`, Якщо ми спочатку замінимо символи `'0'` на `'1'`, то рядок не буде містити нулів і відповідно, коли ми потім навпаки замінимо `'1'` на `'0'`, то рядок не буде містити одиниць. Отже, напишемо програму, що реалізує поставлену задачу здійснюючи заміну для кожного окремого символа рядка.

Нагадаємо, рядок належить до незмінюваних типів, тому ми не можемо безпосередньо змінювати символи у рядку. У цій задачі можна застосувати підхід, що використовувався у Прикладі 5.8. Проте застосуємо інший підхід, а саме перетворимо рядок у список символів, проведемо необхідну заміну у отриманому списку символів і склеїмо символи до купи використовуючи операцію `join`.

Заміну будемо здійснювати таким чином: будемо пробігати список символів і якщо поточний символ буде `'0'` або `'1'`, то будемо змінювати його на `'1'` або `'0'` відповідно.

Отже, програма буде мати вигляд

```
S = input("Введіть рядок ")
l = list(S) # l - список символів рядка S
for i in range(len(l)):
    if l[i] == '0':
        l[i] = '1'
    elif l[i] == '1':
        l[i] = '0'

S = "".join(l)

print(S)
```

Порівняння рядків

Рядки у Python можна порівнювати. Порівняння рядків проводиться лексикографічно.

Будемо казати, що послідовність $a = \{a_1 a_2 \dots a_n\}$ **лексикографічно менша** (або просто менша) за послідовність $b = \{b_1 b_2 \dots b_n\}$, якщо існує таке натуральне k, що для всіх $i \le k$ виконується рівність $a_i = b_i$, а для $i = k + 1$ справедлива нерівність $a_{k+1} < b_{k+1}$. Якщо для всіх $i \le n$ виконується рівність $a_i = b_i$, то будемо казати, що послідовності a і b **лексикографічно рівні** (або просто рівні).

Наприклад, послідовність $\{012\}$ менша за послідовність$\{021\}$. Також якщо згадати, що символи можна порівнювати, порівнюючи їхні номери в алфавіті, то послідовність символів $\{aaa\}$ менша за послідовність $\{aba\}$.

Вищенаведене означення можна узагальнити на випадок послідовностей, у які входить неоднакова кількість членів. Для цього будемо вважати, що порожній член послідовності (тобто його відсутність) менший за будь-який наявний. Отже, з цього можемо зробити висновок, що, наприклад, послідовність символів $\{aaa\}$ менша за послідовність символів $\{abcd\}$.

Лексикографічний порядок послідовностей – це порядок, при якому послідовності відсортовані за (лексикографічним) зростанням.

Прикладом лексикографічного порядку є послідовність слів у словнику.

Для порівняння рядків, у Python як і для символів визначені 6 стандартних відношень

```
==, !=, >, <, >=, <=
```

```
>>> "Pentium Pro" >= "Pentium"
True
>>> "True" <= "False"
False
```

Приклад 5.14. Перевірити чи слова, що записані у рядку знаходяться у словниковому порядку.

Розв'язок. Для розв'язання цієї задачі достатньо розбити рядок на слова і далі, скористатися операцією порівняння рядків для слів отриманого списку. Цю задачу можна віднести до задач пошуку. Дійсно, припустимо спочатку, що слова у рядку розташовані у порядку зростання. Далі будемо шукати послідовну пару слів, таку, що перше слово більше за друге. Якщо таку пару ми знайдемо, то слова у рядку не впорядковані. Отже програма буде мати вигляд

```
S = input("Введіть рядок ")
l = S.split()
ok = True
for i in range(1, len(l)):
    if l[i - 1] > l[i]:
        ok = False
        break

if ok:
    print("Слова в алфавітному порядку")
else:
    print("Слова НЕ в алфавітному порядку")
```

Форматування рядків

Досить часто виникає ситуація, коли потрібно створити рядок, у яких підставляються деякі дані, отримані у процесі виконання програми. Одним зі способів такої підстановки, а саме використання оператора %, ми вже неодноразово користувалися. Недоліком цієї операціє є те, що необхідно наперед знати типи даних, які очікуються для підстановки. Наприклад, у падку використання такої інструкції

```
    print("%c - велика літера" % Ch)
```

Python очікує для підстановки саме символьний літерал у змінній Ch. У випадку, якщо ж тип змінної наперед не відомий, то результат виконання інструкції може бути неочікуваним. Звичайно, можна перед підстановкою

робити зведення літералу до конкретного типу, наприклад, вищенаведену інструкцію змінити на таку

```
    print("%s – велика літера" % str(Ch))
```

проте, зручнішим способом подолання згаданої проблеми є використання методу `format()`.

Метод `format()` подібний до форматування рядка за допомогою оператора `%`, тобто він також задає спосіб підстановки відповідних значень у рядок у відповідні позиції. Проте він має інший синтаксис і не вимагає вказування типів даних, що підставляються.

Розглянемо його синтаксис у найпростішому випадку.

Нехай задано рядок `s`, що містить послідовності символів `{}` (послідовність символів, що складається з відкритої та закритої фігурної дужки), наприклад

```
>>> s = "Мій улюблений предмет {}"
```

Метод формат, застосований до рядка, підставляє аргументи у рядок замість позицій виділених фігурними дужками `{}`.

```
>>> s.format("програмування")
'Мій улюблений предмет програмування'
```

Кількість підстановок у рядок може бути довільна, головне, щоб кількість аргументів дорівнювала кількості підстановок виділених з допомогою `{}` у рядку до якого застосовується метод `format()`.

```
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
```

При задаванні рядка, до якого буде застосовуватися метод `format()`, у фігурних дужках може вказуватися який за номером аргумент буде підставлено. У цьому випадку кількість аргументів методу `format()` може бути меншою за кількість підстановок рядка

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{0}-{1}{0}'.format('abra', 'cad')
'abra-cadabra'
```

Можливості форматування рядків за допомогою оператора % або методу format не обмежуються можливостями описаними у цьому посібнику. Зокрема, під час їхнього використання, можна вказувати точність для дійсних чисел, кількість символів виділених для підстановки, вирівнювання рядка, тощо. Для детальнішого ознайомлення рекомендуємо звернутися до офіційної документації [1] або іншої додаткової літератури.

Задачі для самостійної роботи

5.1. Вивести на екран таку таблицю символів:

```
Aa Bb Cc Dd Ee Ff Gg Hh
Ii Jj Kk Ll Mm Nn Oo Pp
Qq Rr Ss Tt Uu Vv Ww Xx
```

5.2. Визначити, яка з двох заданих літер у даному тексті трапляється частіше.

5.3. Визначити, чи входить до даного тексту

a) кожна з літер слова "`key`";

b) кожна з літер заданого слова.

5.4. Скласти програму підрахунку загального числа входжень символів основних арифметичних операцій (`'+'`, `'-'`, `'*'`, `'/'`) у рядку.

5.5. Дано рядок, що містить принаймні одну цифру. Знайти

a) максимальну цифру, що міститься у рядку;

b) цифру, що входить до рядка найбільшу кількість разів;

c) суму цифр, що входять до рядка.

d) цифри, що трапляються у рядку рівно два рази;

e) цифри, що не міститься у рядку;

5.6. Визначити, кількість різних символів у рядку.

5.7. Знайти символ, який входить у текст найбільшу кількість разів.

5.8. Визначити, чи є заданий текст правильним записом цілого числа (можливо, зі знаком).

5.9. Перевірити, чи є даний рядок ідентифікатором, натуральним числом, чи ні тим ні іншим.

5.10. Задано рядок, що містить електронну адресу користувача. Переконатися, що ця адреса є коректною (наявність символа @ і крапки, а також наявність принаймні двох символів після останньої крапки)

5.11. У заданий текст входять тільки цифри та літери. Визначити, чи задовольняє він такі властивості:

a) текст є десятковим записом числа, кратного 9 (6, 4);

b) текст починається з деякої ненульової цифри, за якою стоять тільки літери, і їхня кількість дорівнює числовому значенню цієї цифри;

c) текст містить (крім літер) тільки одну цифру, причому її числове значення дорівнює довжині тексту;

d) сума парних числових значень цифр, які входять у текст, дорівнює

довжині тексту;

e) текст збігається з початковим (кінцевим, будь-яким) відрізком ряду 0123456789;

f) текст складається тільки з цифр, причому їхні числові значення складають арифметичну прогресію (наприклад, 3 5 7 9, 8 5 2, 2).

5.12. Перетворити заданий текст у такий спосіб:

a) виключити всі цифри та подвоївши знаки `'+'` та `'-'`;

b) виключити всі знаки `'+'`, безпосередньо за якими стоїть цифра;

c) виключити всі літери a, безпосередньо перед якими стоїть літера c;

d) замінити всі пари `"ph"` на літеру `'f'`;

e) замінити кожний символ, що йде за s, заданим символом c.

f) виключити всі зайві пропуски (тобто з кількох, що йдуть підряд, залишити один);

g) виключити групи символів, які стоять між символами `'('` та `')'`. Самі дужки теж виключити. Вважати, що дужки розставлено правильно (парами) та всередині кожної пари дужок немає інших дужок.

h) замінити всі знаки оклику `'!'` знаками питання `'?'`, а кожен знак питання `'?'` – двома знаками `"??"`.

i) видалити кожний символ `'*'` та подвоїти кожний символ, відмінний від `'*'`.

j) видалити всі коми, які передують першій крапці, та замінити знаком `'+'` усі цифри `'3'`, які трапляються після першої крапки.

5.13. Дано рядок, що містить принаймні одну цифру. Написати програму, яка:

a) починаючи із заданої позиції, збільшує всі парні цифри, що містяться у рядку, на 1.

b) всі цифри, що містяться у рядку, замінює їх квадратами.

5.14. Дано рядок. Створити програми, що видаляють з нього

a) символи, що стоять на парних місцях;

b) символи, що дорівнюють символу c;

c) цифри;

d) цифри, що трапляються у рядку більше одного разу;

e) символи, що трапляються у рядку рівно два рази ;

f) символи, що не є цифрами;

g) символи, що належать відрізку $[c1, c2]$.

h) симетричні початок та кінець. Наприклад, рядок `"abcdefba"` має перетворитися у рядок `"cdef"`.

5.15. Замінити всі пари однакових символів рядка, які йдуть підряд, одним символом. Наприклад, рядок `"aabcbb"` має перетворитися на рядок `"abcb"`.

5.16. Скласти процедуру "стискання" рядка: кожний підрядок, який складається з кількох входжень деякого символа, замінюється на цей символ.

5.17. Видалити з рядка всі повторні входження символів.

5.18. Побудувати рядок S із рядків $S1$, $S2$ так, щоб у S входили

i) ті символи $S1$, які не входять у $S2$;

j) усі символи $S1$, які не входять у $S2$, та всі символи $S2$, які не входять у $S1$.

5.19. Скласти підпрограму, яка з першого заданого рядка видаляє кожний символ, який належить і другому заданому рядку.

5.20. Задане натуральне число n. Перетворити його до системи числення з основою b ($2 < b < 16$).

5.21. Знайти у даному рядку символ та довжину найдовшої послідовності однакових символів, що йдуть підряд.

5.22. Скласти програму підрахунку найбільшої кількості цифр, що йдуть підряд у рядку.

5.23. Заданий рядок, який складається з великих літер латинського алфавіту. Скласти програму перевірки впорядкованості цих літер за алфавітом.

5.24. Рядок називається монотонним, якщо він складається зі зростаючої або спадної послідовності символів. Скласти програму перевірки монотонності рядка.

5.25. Виділити з рядка найбільший монотонний підрядок, коди послідовних символів якого відрізняються на 1.

5.26. Дано число n з проміжку від 0 до 10^9. Надрукувати його українськими словами (наприклад, 234 = двісті тридцять чотири).

5.27. Скласти програму виведення на друк усіх цифр, які входять у заданий рядок, та окремо – решту символів, зберігаючи при цьому взаємне розташування символів у кожній із цих двох груп.

5.28. Скласти програму виведення на друк тільки маленьких літер латинського (українського) алфавіту, які входять у заданий рядок.

5.29. Вивести на друк усі символи рядка, що складається з маленьких літер латинського алфавіту, відповідними великими літерами.

5.30. Скласти програму виведення на друк в алфавітному порядку всіх різних маленьких латинських (кириличних) літер, які входять до даного рядка.

Текстом будемо називати послідовність символів (рядок), що містить лише символи латинського (та/або українського) алфавіту, цифри, символ пропуску, та знаки пунктуації. **Словом** будемо називати групу символів, що складається лише з символів латинського (та/або українського) алфавіту та цифр і обмежена зліва та справа одним чи кількома пропусками, знаками пунктуації або початком чи кінцем тексту.

5.31. Заданий текст, що містить послідовність слів. Визначити кількість слів, які:

a) містяться в тексті;

b) починаються із заданої літери;

c) закінчуються заданою літерою;

d) починаються і закінчуються заданою літерою;

e) починаються і закінчуються однією літерою;
f) містять принаймні одну задану літеру;
g) містять рівно три заданих літери.

5.32. Заданий текст, що містить послідовність слів. Знайти
a) найкоротше слово рядка, його довжину та позицію у рядку;
b) найдовше слово рядка, його довжину та позицію у рядку;
c) кількість входжень заданої літери в n-те, починаючи з початку слово даної послідовності;
d) слово, що містить найбільшу кількість голосних літер, кількість голосних літер у цьому слові та позицію цього слова у тексті;
e) всі слова, які є натуральними числами;
f) всі слова, які є паліндромами (симетричними);
g) всі слова, які є ідентифікаторами;

5.33. Заданий текст, що містить послідовність слів. Описати програми, що за заданим текстом будують новий рядок у якому:
a) записані усі слова вихідного тексту у зворотному порядку;
b) записані усі слова які трапляються у вихідному тексті по одному разу;
c) усі слова, вихідного рядка записані без повторів;
d) усі слова вихідного тексту записані у порядку зростання;

5.34. Дано два рядки, кожен з яких містить послідовність слів. Описати програми, що друкують:
a) усі слова першого рядка, що не входять до другого рядка;
b) усі різні слова першого рядка, що входять до другого рядка;
c) усі різні слова, що входять до обох рядків;
d) усі слова, що входять до обох рядків тільки по одному разу;
e) усі різні слова, що містяться тільки в одному з рядків.

5.35. Дано текст, що має вигляд:

$$d_1 \pm d_2 \pm \ldots \pm d_n$$

де $\mathrm{d_i}$, $1 \le i \le n$ – цифри, $n > 1$. Визначити записану в тексті алгебраїчну суму.

5.36. Дано текст, що є правильним записам алгебраїчного виразу котрий складається з чисел, знаків арифметичних операцій та дужок (для зміни пріоритету арифметичних операцій), наприклад

$$(2 + 2) * 2$$

Обчислити значення цього виразу.

5.37. Дано рядок, що містить цифри та символи `x, *, +, -, =` і є правильним записом квадратного рівняння. Знайти корені цього рівняння.

5.38. Дано рядок, що містить букви та дужки кількох видів: `'('`, `')'`, `'['`, `']'`, `'{'`, `'}'`. Перевірити, чи правильно в заданому виразі розставлені дужки.

5.39. Як відомо, римська система числення використовує символи M, D, C, L, X, V та I для чисел 1000, 500, 100, 50, 10, 5 та 1 відповідно. Інші числа записуються використовуючи різні комбінації вищенаведених символів, наприклад

```
I - 1          VII - 7         XLVI - 46       CCCII - 302
II - 2         VIII - 8        L - 50          CDXLI - 441
III - 3        IX - 9          LXXV - 75       ID - 499
IV - 4         X - 10          XCII - 92       D - 500
V - 5          XVIII - 18      IC - 99         DCXCV - 695
VI - 6         XXXI - 31       C - 100         CM - 900
```

Написати програму, що перетворює число записане у римській системі числення у десяткову і навпаки.

5.40. Дано рядок, що містить дріб чисельник і знаменник якого записані у римській системі числення. Необхідно скоротити цей дріб, записаний у римській системі числення.

5.41. Дано два натуральних числа записаних у римській системі числення. Знайти їхню суму, різницю та добуток. Результат має бути також записаним у римській системі числення.

5.42. Дано текст, що містить натуральні числа, операції відношень та логічні операції, таким чином, що цей текст є правильним записом деякого логічного виразу (умови). Наприклад:

$$1 > 3 \text{ and } 3 == 5$$

Обчислити значення цього логічного виразу.

§6 СЛОВНИКИ ТА МНОЖИНИ

Невпорядковані колекції

Розглянути раніше колекції (список, кортеж, рядок) були впорядкованими. Всі елементи у колекції займали свою чітко визначену позицію. До елементів цих типів можна здійснювати доступ за індексами.

Крім впорядкованих типів даних у Python існують невпорядковані колекції, у яких не визначено порядок (послідовність) доступу до елементів. До таких типів відносяться словники та множини.

Словники

Словник у Python – це невпорядкована колекція об'єктів довільних типів з доступом по ключу.

Ключем словника може бути об'єкт незмінюваного типу (число, рядок, кортеж), за допомогою якого можна однозначно звернутися до елемента словника. Словники називають асоціативними масивами або хеш-мапами. Словники у Python належать до **змінюваних** (**mutable**) типів. Значення словника зберігаються в невідсортованому порядку, більш того, ключі можуть зберігатися не в тому порядку, в якому вони додаються до колекції.

Для прикладу розглянемо словник студентів.

Ключ має однозначно ідентифікувати студента, тому у ролі ключа можуть бути лише унікальні для студента характеристики. Наприклад, такими характеристиками можуть бути **номер студентського квитка** або **ідентифікаційний код**. Прізвище та Ім'я не можуть бути ключами, оскільки у одній групі може існувати кілька осіб з однаковим прізвищем та ім'ям.

```
d = {"AA333333" : "Іваненко Іван",
        ... ,
     "BB123123" : "Петренко Петро",
        ... ,
     "AA999999" : "Петренко Петро" }
```

Створення словників

1. За допомогою літерала. Пари ключ та значення елементів словника записуються через двокрапку. Різні пари розділяються комою, а вся послідовність записується у фігурних дужках:

```
empty_dict = {}          # Порожній словник

d = {'1': 1, 2 : '2'}    # Словник з 2-х пар ключ-значення
                         # ключу '1' відповідає значення 1
                         # ключу 2 відповідає значення '2'

# Словник ключами якого є кортежі (1, 1) та (2, 2),
# а значеннями відповідно два списки
d1 = {(1, 1) : [1, 2, 3], (2, 2) : [4, 5, 6]}
```

2. За допомогою перетворення у словник іншої колекції використовуючи ключове слово `dict`.

```
d = dict(collection)
```

Тут `d` – новий словник, `collection` – колекція у якій записані пари ключ-значення.

```
>>> d = dict(short='dict', long='dictionary')
>>> print(d)
{'short': 'dict', 'long': 'dictionary'}
>>> d = dict([(1, 1), [333, 334], (2, 4)])
>>> print(d)
{1: 1, 2: 4, 333: 334}
```

3. За допомогою методу `fromkeys`.

```
d = dict.fromkeys(collection, initial_value)
```

Тут `collection` – колекція ключів, `initial_value` – не обов'язковий параметр, що задає початкове значення, яке відповідає всім ключам

```
>>> d = dict.fromkeys(['a', 'b'])
>>> print(d)
{'b': None, 'a': None}
>>> d = dict.fromkeys(['a', 'b'], [10, 19])
>>> print(d)
{'b': [10, 19], 'a': [10, 19]}
```

4. За допомогою оператор створення словника (словникоутворення).

Оператор створення словника – це спосіб побудови нового словника на базі колекції, до всіх елементів якої застосовується деякий вираз:

```
D = {key(i) : expr(i) for i in collection if condition}
```

Оператор працює таким чином: змінна `i` послідовно пробігає всі елементи `collection`. Якщо для поточного значення `i` виконується умова `condition`, то в словник `D` додається пара `key(i) : expr(i)`.

```
>>> d = {i // 2: i**2 for i in range(10) if i % 2 == 0}
>>> print(d)
{4: 64, 0: 0, 2: 16, 1: 4, 3: 36}
```

Доступ до елементів словника

Як було сказано раніше, словник це набір елементів з доступом за ключем. Власне ключ словника, це аналог індексу для списку. Для того, щоб звернутися до відповідного елемента словника, необхідно вказати ключ цього елементу у квадратних дужках:

```
>>> d = {'first' : 1, 'second' : 2}
>>> d['first']
1
>>> d['third']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'third'
```

Як видно з прикладу, звернення до словника за неіснуючим ключем породжує помилку.

Присвоєння по існуючому ключу перезаписує значення елемента, що відповідає цьому ключу, а присвоєння по новому ключу, розширяє словник новою парою ключ-значення

```
>>> d = {'first': 1, 'second': 2}
>>> d['first'] = 111
>>> d
{'first': 111, 'second': 2}
>>> d['third'] = 3
>>> d
{'first': 111, 'second': 2, 'third': 3}
```

Для видалення пари зі словника, використовують оператор **`del`**

```
>>> d = {'first': 1, 'second': 2}
>>> del d['first']
```

```
>>> d
{'second': 2}
```

Операції над словниками

Нехай `d` деякий заданий словник. Розглянемо основні операції, що можна здійснювати зі словником.

Таблиця 6.1. Перевірка приналежності ключа або значення до словника

Операція	Опис
`key in d`	Повертає **`True`**, якщо ключ `key` входить до `d`
`key not in d`	Повертає **`True`**, якщо ключ `key` не входить до `d`
`v in d.values()`	Повертає **`True`**, якщо значення `v` входить до `d`
`v not in d.values()`	Повертає **`True`**, якщо значення `v` не входить до `d`

```
>>> d = {'first': 1, 'second': 2}
>>> 'first' in d
True
>>> 'third' in d
False
>>> 222 in d.values()
False
>>> 2 in d.values()
True
```

Таблиця 6.2. Вибір елементів чи колекцій словника

Операція	Опис
`d.get(k)`	Повертає значення ключа `k` або **`None`**, якщо ключа `k` немає у словнику.
`d.get(k,v)`	Повертає значення ключа `k` або `v`, якщо ключа `k` немає у словнику.
`d.items()`	Повертає колекцію всіх пар ключ-значення в словнику `d`.
`d.keys()`	Повертає колекцію всіх ключів словника `d`.
`d.values()`	Повертає колекцію всіх значень в словнику `d`.

```
>>> d = {'first': 1, 'second': 2}
>>> d.get('first')
1
>>> d.get("third", 3)
3
```

```
>>> d.items()
dict_items([('first', 1), ('second', 2)])
>>> d.values()
dict_values([1, 2])
```

Таблиця 6.3. Аналіз словника

Операція	Опис
`len(d)`	Довжина `d` (кількість пар ключ-значення)
`min(d)`	Найменший ключ словника `d` (якщо ключі можна порівнювати)
`max(d)`	Найбільший ключ словника `d` (якщо ключі можна порівнювати)

```
>>> d = {'first': 1, 'second': 2}
>>> len(d)
2
>>> min(d)
'first'
```

Вищенаведені операції можна проводити не лише з ключами, але і з значеннями словника. Для цього потрібно замість словника брати колекцію його значень, наприклад, інструкція

```
min(d.values())
```

знайде найменше значення, що міститься у словнику.

Таблиця 6.4. Модифікація словника.

Операція	Опис
`d.clear()`	Видаляє всі елементи зі словника `d`
`d.copy()`	Повертає копію словника `d`
`d.pop(k)`	Повертає значення ключа `k` і видаляє зі словника елемент з ключем `k` (якщо ключа `k` немає у словнику, то видає помилку)
`d.pop(k,v)`	Повертає значення ключа `k` і видаляє зі словника елемент з ключем `k`. Якщо ключа `k` немає у словнику, то повертає значення v
`d.popitem()`	Повертає і видаляє довільну пару ключ-значення зі словника `d` (якщо словник `d` порожній, то генерує помилку)
`d.update(d1)`	Додає в словник `d` пари ключ-значення з колекції `d1`, які відсутні в словнику `d`, а для кожного ключа, який вже присутній в словнику `d`, виконує заміну відповідним значенням з `d1` (колекція `d1` може бути як словником, так і будь-якою колекцією, що містить пари ключ-значення).

```
>>> d = {'first': 1, 'second': 2}
>>> d.pop('first')
1
>>> d
{'second': 2}
>>> d = {'first': 1, 'second': 2}
>>> d1  = (('third', 3), ('forth', 4), ('first', 111))
>>> d.update(d1)
>>> d
{'forth': 4, 'third': 3, 'first': 111, 'second': 2}
>>> d.clear()
>>> d
{}
```

Обхід словників

Нехай `d` заданий словник. Розглянемо основні види обходу словника
1. Обхід словника за ключами

```
for key in d:
    process_iteration
```

Так, наприклад, результат виконання коду

```
>>> d = {'first': 1, 'second': 2, 'third': 3}
>>> for key in d:
        print(key)
```

стане виведення наступного тексту

```
second
third
first
```

2. Обхід словника за значеннями

```
for val in d.values():
    process_iteration
```

```
>>> d = {'first': 1, 'second': 2, 'third': 3}
>>> for val in d.values():
        print(val)

2
3
1
```

3. Обхід словника за парами ключ-значення

```
for key, val in d.items():
    process_iteration
```

```
>>> d = {'first': 1, 'second': 2, 'third': 3}
>>> for key, val in d.items():
        print(key, val)

second 2
third 3
first 1
```

Приклад 6.1. Слова у рядку розділяються одним або декількома пропусками. Визначити кількість входжень кожного слова до рядка.

Розв'язок. Для розв'язання задачі побудуємо на базі рядка словник, у якому ключами будуть унікальні слова, а значеннями кількість входжень слова до рядка.

```
S = input("введіть рядок: ")
words_list = S.split() # розбиваємо рядок на список слів
d = {}                 # порожній словник слів
for word in words_list: # пробігаємо по всіх словах
    if word not in d:   # якщо слово не входить до списку
                        # додаємо його до словника разом з
                        # його кількістю входжень
        d[word] = words_list.count(word)

print('слова та кількість входжень:', d)
```

Приклад 6.2. Визначити слово, яке входить до тексту найбільшу кількість разів.

Розв'язок. У цій задачі можемо використати результат отриманий у прикладі 6.1**.** Припустимо, що словник слів прикладу 6.1 вже побудований. Пробіжимо по словнику та знайдемо ключ, якому відповідає найбільше значення.

```
m = 0          # m - максимальна кількість входжень
max_word = '' # max_word - змінна що містить слово,
               # яке входить найбільшу кількість разів
for word, count in d.items(): # пробігаємо словник по
                              # всіх парах ключ-значення
    if m < count:
        m = count
        max_word = word

print('слово, яке входить найчастіше -', max_word)
```

Множини

Множина у Python – це невпорядкована колекція унікальних об'єктів (тобто таких, що не повторюються) довільних типів.

Множини у Python належать до змінюваних (mutable) типів. Елементи зберігаються в невідсортованому порядку та не залежать від того коли вони додаються у множину.

Множини Python використовуються для розв'язання задач у яких має значення лише приналежність елемента до деякої множини або відсутність елемента у множині. Робота з множинами у Python подібна до роботи з дискретними множинами у математиці.

Створення множин

1. За допомогою літерала. Елементи множини розділяються комою, а вся послідовність записується у фігурних дужках, аналогічно словникам.

```
empty_set = set()    # Порожня множина може створюватися
                     # лише з допомогою інструкції set

M = {1, 2, 3, 3, 3}    # Множина {1, 2, 3}
```

Зверніть увагу на створення порожньої множини. Нагадаємо, що порожні фігурні дужки використовуються для створення словників. Тому, щоб уникнути неоднозначності порожні множини створюються з використанням функції `set`.

2. За допомогою перетворення у множину іншої колекції використовуючи ключове слово `set`.

```
M = set(collection)
```

Тут `M` – нова множна, що будується на базі колекції `collection`.

```
>>> M = set('мама')
>>> print(M)
{'м', 'а'}
```

3. За допомогою оператор створення множини.

Оператор створення множини – це спосіб побудови множини на базі колекції, до всіх елементів якої застосовується деякий вираз:

```
M = {expr(i) for i in collection if condition}
```

Цей оператор працює таким чином: змінна `i` послідовно пробігає всі елементи `collection`. Якщо для поточного значення `i` виконується умова `condition`, то в множину `M` додається елемент `expr(i)`.

```
>>> M = {i for i in "Boing 777" if i.isdigit()}
>>> print(M)
{'7'}
```

Операції над множинами

Нехай `M` деяка задана множина. Розглянемо основні операції, що можна здійснювати з множиною.

Таблиця 6.5. Операції відношення.

Операція	Опис
`x in M`	Повертає **`True`**, якщо елемент `x` входить до `M`
`x not in M`	Повертає **`True`**, якщо елемент `x` входить до `M`
`M == N`	Повертає **`True`**, якщо множини `M` і `N` складаються з однакових елементів
`M != N`	Повертає **`True`**, якщо множини `M` і `N` не рівні
`M < N`	Повертає **`True`**, якщо множина `M` є строго підмножиною `N`
`M <= N`	Повертає **`True`**, якщо множина `M` є підмножиною `N` або збігається з `N`.
`M > N`	Повертає **`True`**, якщо множина `N` є строго підмножиною `M`
`M >= N`	Повертає **`True`**, якщо множина `N` є підмножиною `N` або

Операція	Опис
	збігається з M.

```
>>> M = {1, 2, 3, 444}
>>> 444 in M
True
>>> 222 in M
False
>>> 222 not in M
True
>>> N = {1, 2, 3, 444}
>>> M == N
True
>>> M < N
False
>>> M <= N
True
>>> M = {1, 2}
>>> N = {2, 3, 4}
>>> M <= N
False
```

Таблиця 6.6. Перетин, об'єднання, різниця.

Операція	Альтернативна операція	Опис
M | N	M.union(N)	Об’єднання множин M U N
M & N	M.intersection(N)	Перетин множин M ∩ N
M - N	M.difference(N)	Різниця множин M \ N
M ^ N	M.symmetric_difference(N)	Симетрична різниця множин M та N

```
>>> M = {1, 2, 3}
>>> N = {2, 3, 4, 5}
>>> M | N
{1, 2, 3, 4, 5}
>>> M & N
{2, 3}
>>> M.intersection({3, 4})
{3}
>>> N - M
{4, 5}
>>> N ^ M
{1, 4, 5}
```

Таблиця 6.7. Додавання та видалення елементів.

Операція	Опис
`M.add(x)`	Додає елемент `x` до множини `M`
`M.discard(x)`	Видаляє елемент `x` з множини `M`, якщо цей елемент є у множині.
`M.remove(x)`	Видаляє елемент `x` з множини `M`. Породжує помилку, якщо елемент `x` відсутній.
`M.pop()`	Повертає довільний елемент що входить до множини `M` та видаляє його з множини `M`. Породжує помилку, якщо множина порожня.
`M.clear()`	Видаляє всі елементи з множини `M`
`M.update(N)`	Оновлює `M` значенням об'єднання `M` та `N` (тобто виконує інструкцію `M = M | N`)

```
>>> M = {1, 2, 3}
>>> M.add(444)
>>> M
{1, 2, 3, 444}
>>> M.remove(444)
>>> M
{1, 2, 3}
>>> M.discard(444)
>>> M
{1, 2, 3}
>>> M.pop()
2
>>> M
{1, 3}
```

Приклад 6.3. Наведемо розв'язання задачі прикладу 6.1 використовуючи операції із множинами.

Розв'язок. Як і в прикладі 6.1 побудуємо словник, що містить слова у якості ключів та число входжень як значення. Щоб не перевіряти, чи міститься вже слово в словнику, побудуємо множину слів, що містяться у тексті.

```
S = input("введіть рядок: ")

words = S.split()           # список всіх слів
words_set = set(words)      # створюємо множину слів
d = {}                      # порожній словник слів

for word in words_set:      # пробігаємо по всіх словах
    d[word] = words.count(word)
```

```
print('слова та кількість входжень', d)
```

Незмінні множини

Окрім звичайних множин, у Python існують захищені від змін множини (незмінні множини) `frozenset`.

Незмінні множини `frozenset` належать до типу immutable**.** Фактично, різниця між множиною і незмінною множиною схожа на різницю між списками і кортежами: над незмінними множинами можна проводити всі операції, що і над звичайними множинами, що не змінюють їх. `frozenset` використовують тоді, коли множина після створення не змінюється і потрібна більша швидкодія у порівнянні з використанням звичайних множин.

Створюються незмінні множини перетворенням будь-якої колекції за допомогою інструкції `frozenset`

```
>>> M = frozenset({1, 2, 3})
>>> M
frozenset({1, 2, 3})
```

`frozenset` можуть фігурувати у виразах разом із звичайними множинами. При цьому результат виразу буде того типу, до якого належить перший операнд операції (`set` або `frozenset`).

```
>>> M = frozenset({1, 2, 3})
>>> N = {1, 3, 2}
>>> M == N
True
```

```
>>> M = frozenset({1, 2, 3})
>>> N = {3, 4}
>>> M | N
frozenset({1, 2, 3, 4})
>>> N | M
{1, 2, 3, 4}
```

Задачі для самостійної роботи

6.1. Задано дві послідовності натуральних чисел. Визначте скільки чисел міститься як у першому, так і другому списках. Виведіть ці числа у порядку зростання.

6.2. Кілька людей грає в гру «міста». Реалізуйте програму, що буде контролювати, чи назване місто не використовувалося раніше.

6.3. В Інститут філології університету кожен зі студентів вивчає принаймні чотири іноземних мови. Дано словник студентів інституту філології у якому у якості значення міститься список мов, які вивчає студент. Визначте:

a) всі мови, що вивчаються у інституті філології;
b) мови, які вивчають всі студенти інституту;
c) мови, які вивчає переважна більшість;
d) мови, які вивчає менше четверті студентів.

6.4. Задано послідовність натуральних чисел. Перевірити чи правда, що

a) всі задані числа є різними;
b) всі задані числа є простими;
c) всі задані числа є простими і різними;
d) серед тих чисел, що є не простими всі парні;

6.5. Задано послідовність чисел з діапазону $[1, N]$. Вибрати ті з них, що є числами Фібоначчі. При цьому

a) визначити загальну кількість таких чисел;
b) визначити кількість різних серед цих чисел;
c) вказати їхні номери у послідовності чисел Фібоначчі відсортовані за зростанням;

6.6. Дано натуральне число.

a) Визначити зі скількох різних цифр складається десятковий запис цього числа.
b) Надрукувати у порядку зростання всі цифри, які не входять до десяткового запису цього числа.

6.7. Задано рядок, що містить правильний запис деякого арифметичного виразу. Перевірити чи правда, що рядок є

a) сумою натуральних чисел;
b) добутком дійсних чисел;
c) раціональним дробом.

6.8. Задано рядок, що складається з латинських (українських) літер. Надрукувати всі літери, що

a) входять до рядка по одному разу;
b) входять до рядка не менше двох раз;
c) не входять до рядка.

6.9. Слова у рядку розділяються одним або декількома пропусками. Знайти

a) частоту вживання кожного слова, тобто відношення кількості входження слова до загального числа всіх різних слів;
b) слова, що входять до рядка не менше двох разів;

c) слова з n букв, що входять до рядка принаймні m разів.

6.10. Дано список країн та міст, що знаходяться у кожній з заданих країн. Для заданого міста знайдіть країну у якій воно знаходиться.

6.11. Відомості про автомобіль складаються з його марки, унікального номера і прізвища власника. Дано словник, який містить відомості про кілька автомобілів. Скласти програми знаходження

a) прізвищ власників номерів автомобілів даної марки;
b) кількості автомобілів кожної марки.

6.12. Нехай словник моделює стан гаманця у такий спосіб:
значення банкноти (номінал) – це ключ, а кількість банкнот цього номіналу – значення.
якщо у гаманці відсутня банкнота певного номіналу, то такий елемент відсутній у гаманці (не допускається у словнику наявність пар типу `{20:0}`)

Потрібно порахувати суму коштів, що міститься у гаманці. Вважається, що повний перелік номіналів банкнот відомий.

6.13. Відомі дані про масу й об'єм N предметів, виготовлених із різних матеріалів. Знайти предмет, густина матеріалу якого найбільша.

6.14. Відомі дані про чисельність населення (у мільйонах жителів) та площі N держав. Знайти країну з мінімальною щільністю населення.

6.15. Відомо кількість очок, отриманих кожною з N команд, що взяли участь у чемпіонаті з футболу. Жодна пара команд не набрала однакової кількості очок. Визначити:

a) команду, що стала чемпіоном;
b) команди, що посіли друге й третє місця;
c) команду, що посіла останнє місце.

6.16. Задано англо-український словник. Для кожного англійського слова у цьому словнику може бути декілька перекладів – синонімів. Написати програму, що по заданому словнику будує україно-англійський словник.

6.17. Дано словник, що у якості ключів містить назви пір року (зима, весна, літо, осінь), а у ролі значень – кортеж з назв місяців року, що належать до цієї пори року. Здано таблицю, що задається словником з ключами номер місяця у році, і значеннями – показниками середньомісячних опадів у деякому регіоні. Знайти:

a) пору року на яку припадає найбільша кількість опадів;
b) найсухішу пору року;
c) пору року з найбільшими середньомісячними коливаннями кількості опадів;

6.18. Використовуючи словники запропонувати та реалізувати метод шифрування заданого рядка символів та дешифрування його до вихідного рядка.

6.19. Реалізувати метод шифрування послідовності символів, використовуючи для співставлення значення-шифр

a) послідовність чисел Фібоначчі;

b) послідовність чисел трибоначчі;
c) деяку послідовність натуральних чисел, задану деяким рекурентним співвідношення.

6.20. Відомості про студента складаються з його унікального номера (студентського квитка), імені, прізвища, року навчання та переліку оцінок отриманих на останній сесії. Дано словник, який містить відомості про студентів деякого факультету. Скласти програми, які дозволяють

a) визначити чи навчаються на факультеті студенти з однаковими прізвищами;
b) визначити середню успішність студентів на кожному курсі;
c) визначити кількість студентів на кожному курсі, що мають принаймні одну двійку;
d) надрукувати список студентів, що успішно склали сесію тобто які не мають оцінок нижче чотирьох;
e) надрукувати список найкращих студентів, тобто таких, які мають всі відмінні оцінки;

§7 ОБРОБКА ВИКЛЮЧЕНЬ

Помилки, виключні ситуації та виключення

Помилки у програмах

Розглянемо дуже важливий розділ сучасного програмування – обробку помилок.

Існує три типи помилок, що виникають під час виконання програми:

- **синтаксичні** – це помилки, що породжуються неправильним використанням синтаксичних конструкцій мови програмування. Наприклад, такий виклик оператора `print`

```
print("Повідомлення!" # Не вистачає дужки
```

породить таку помилку

```
  File "exception.py", line 4

    ^
SyntaxError: unexpected EOF while parsing
```

- **помилки виконання** (**runtime error**) – це помилки, що виникають у синтаксично правильних програмах під час їхнього виконання, коли інтерпретатор не може коректно розв'язати проблему, що виникла або подальше виконання програми є неможливим. Класичним прикладом помилки виконання є ділення на нуль:

```
>>> div_by_zero = 1 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

Як бачимо, у цьому випадку інтерпретатор повідомив нас про те, що у програмі відбувається ділення на нуль.

- **логічні помилки** – це помилки пов'язані з логікою програми. Інтерпретатор не зможе виявити і ідентифікувати такі помилки, оскільки з точки зору синтаксису програма написана правильно. Їх можна виявити лише за результатами роботи програми.

Існує ціла сфера діяльності – тестування програмного забезпечення (eng: QA (Quality Assurance)) – пов'язана з виявленням помилок останнього типу. Цей же параграф присвячений обробці помилок перших двох типів.

Виключні ситуації та виключення

Якщо у програмі виникає помилка першого або другого типу, то будемо казати, що відбулася **виключна ситуація**. Будь-яка виключна ситуація **генерує** (також використовується термін **породжує**) **виключення**.

Виключення (eng: **exception**) – спеціальний тип даних, що використовується для ідентифікації та детального опису помилки, що виникла у програмі.

Слід зауважити, що виключення породжуються не лише у разі виникнення виключної ситуації, а і як повідомлення про те, що відбулася певна подія. Наприклад, метод списку `pop()` генерує виключення `IndexError`, якщо список порожній.

Обробка виключень

Для обробки виключень використовується оператор **`try`**. Його загальний синтаксис такий

```
try:
    processing_code_with_potential_exception
except exception_1
    handling_exception_1
...
except exception_N
    handling_exception_N
else:
    state_else
finally:
    obligatory_code
```

Виконання цього оператора відбувається таким чином:

1. Інтерпретатор намагається виконати блок коду з потенційним виключенням (`processing_code_with_potential_exception`).
2. Якщо у випадку виконання коду породжується виключення, що належить до класу виключень `exception_1` то у точці генерування виключення відбувається безумовний перехід на блок обробки виключення `handling_exception_1`. Аналогічне стосується інших типів виключення, що перехоплюються у програмі.

3. Якщо, породжене виключення не обробляється у жодному з блоків обробки виключень, то інтерпретатор припиняє роботу програми (за допомогою типового обробника виключень)
4. Якщо виконання блоку **try** повністю відбулося без виключень, то відбувається виконання коду `state_else` блоку **else**. Якщо ж було згенеровано хоча б одне виключення цей блок не виконується.
5. Код `obligatory_code` блоку **finaly** виконується завжди після виконання попередніх кроків незалежно від того, чи було породжено виключення чи ні. Наприклад, без нього не можливо обійтися, якщо необхідно закрити відкритий (до породження виключення) файл, звільнити з'єднання з базою даних тощо.

Слід зауважити, що обов'язковими блоками оператора **try** є лише блок **try** і один блок **except** обробки виключень.

```
try:
    print(1 / 0)           # Блок з потенційним діленням на 0
except ZeroDivisionError: # Обробка виключення ділення на 0
    print("Ділення на 0")
```

або блок **try** і блок **finally**.

```
f = open("myfile.txt")    # Відкриття файлу
try:
    f.read()              # Читання з файлу
finally:
    f.close()             # Закриття файлу
```

Якщо у інструкції **except** опустити клас виключення, то такий блок буде перехоплювати усі виключення. На практиці не рекомендується використовувати такі блоки обробки виключень, оскільки є небезпека перехопити виключення, що є повідомлення системі, а не помилкою або виключення, після яких подальша робота програми не має сенсу.

Приклад 7.1. Задано послідовність дійсних невід'ємних чисел. Знайдемо максимум відношень між сусідніми членами цієї послідовності.

Розв'язок. Очевидно, що якщо деякий член послідовності буде дорівнювати нулю, то в момент знаходження відношення буде генеруватися виключення «ділення на нуль». Очевидно, що просто ігнорувати такі члени послідовності не можна, оскільки результат роботи програми у такому разі буде не об'єктивним.

```
# Введення послідовності
N = int(input("N = "))
```

```
l = []
for i in range(N):
    l.append(float(input("l[%d] = " % i)))
# Знаходження максимуму відношення
try:
    max_ratio = 0
    for i in range(1, len(l)):
        ratio = l[i] / l[i-1]
        if ratio > max_ratio:
            max_ratio = ratio
except ZeroDivisionError:
    print("У послідовності є нульові елементи")
else:
    print("Максимум відношень = %f" % max_ratio)
```

Звернемо увагу на те, що у нашій програмі оператор **try**, у випадку, якщо породжується виключення «ділення на нуль», передає керування блоку **except** тим самим перериваючи роботу циклу **for**, подібно до того, як це здійснює оператор **break**.

Приклад 7.2. Обчислимо суму послідовності цілих чисел, що вводяться користувачем.

Розв'язок. Для введення цілих або дійсних чисел з клавіатури ми користувалися інструкцією `input`, з подальшим перетворенням введеного результату до цілого чи дійсного типу за допомогою інструкцій `int` або `float` відповідно. При цьому, раніше ми вважали, що користувач дисциплінований і вводить з клавіатури лише коректні дані. Проте, якби користувач ввів рядок, який не можливо було б перетворити до відповідного типу, породжувалося виключення `ValueError`.

Напишемо програму, що буде контролювати правильність введення даних. Завершувати підрахунок будемо у випадку, коли користувач введе з клавіатури слово “стоп”.

```
print("Введіть 'стоп' для отримання результату")
suma = 0
while True:
    x = input("Введіть число: ")
    if x == "стоп":
        break  # Завершення підрахунку
    try:
        x = int(x) # Перетворюємо рядок у число
    except ValueError:
        print("Потрібно задати ціле число!")
    else:
```

```
        suma += x
print("Сума = ", suma)
```

Процес роботи програми буде виглядати так

```
Введіть 'стоп' для отримання результату
Введіть число: 1
Введіть число: 2
Введіть число: щось
Потрібно задати ціле число!
Введіть число: 3
Введіть число: стоп
Сума =  6
```

Стандартні класи виключень

Набір вбудованих виключень у мові Python організований у вигляді ієрархії, що наведена нижче.

```
BaseException
      SystemExit
      KeyboardInterrupt
      GeneratorExit
      Exception
            ArithmeticError
                  FloatingPointError, OverflowError,
                  ZeroDivisionError
            AssertionError
            AttributeError
            BufferError
            EOFError
            ImportError
            LookupError
                  IndexError, KeyError
            MemoryError
            NameError
                  UnboundLocalError
            OSError
                  ConnectionError, FileExistsError,
                  FileNotFoundError, InterruptedError,
                  IsADirectoryError, NotADirectoryError,
                  PermissionError, ProcessLookupError,
                  TimeoutError
            ReferenceError
```

```
            RuntimeError
            NotImplementedError
            StopIteration
            SyntaxError
            SystemError
            TypeError
            ValueError
            Warning
```

Якщо читач вже знайомий з парадигмою об'єктно-орієнтованого програмування, то вищенаведена ієрархія може йому нагадати ієрархію класів успадкованих від базового класу `BaseException`. Саме таким чином в Python і організована робота з виключеннями – кожний тип виключення це клас успадкований від іншого типу виключення.

Перевага використання ієрархії для обробки виключень полягає у тому, що є можливість задавання базового класу виключень для перехоплення всіх дочірніх виключень. Так наприклад, у попередніх прикладах для перехоплення виключення «ділення на нуль» використовувалося виключення `ZeroDivisionError`. Якщо ж замість нього вказати базове виключення `ArithmeticError`, то будуть перехоплюватися всі типи виключень, що є нащадками до цього класу: `FloatingPointError`, `OverflowError`, `ZeroDivisionError`.

Нижче наведена таблиця та опис виключень, які будуть зустрічатися у цьому курсі. З описом інших виключень можете ознайомитися за допомогою додаткової літератури, наприклад офіційної документації [1].

Таблиця 7.1. Вбудовані виключення.

Виключення	Опис
`BaseException`	Базовий клас виключень.
`Exception`	Основний клас системних виключень. Всі власні виключення повинні успадковуватися від цього класу.
`ArithmeticError`	Арифметична помилка
`AssertionError`	Помилка твердження – породжується, якщо умова твердження є хибною.
`EOFError`	Породжується функцією зчитування з файлу, якщо досягнуто кінця файлу.
`ImportError`	Помилка імпорту модуля чи пакету
`LookupError`	Помилка індексу або ключа при зверненні до елементів колекції.
`NameError`	Не знайдено змінної з таким ім'ям.

Виключення	Опис
`OSError`	Клас виключень, пов'язаних з операційною системою.
`RuntimeError`	Породжується, коли виключення не потрапляє у жодну з категорій.
`StopIteration`	Породжується вбудованою функцією `next`, якщо в колекції вже перебрані послідовно всі елементи.
`SyntaxError`	Синтаксична помилка
`SystemError`	Внутрішня системна помилка
`TypeError`	Помилка типу – породжується якщо операція застосовується до об'єкта невідповідного типу
`ValueError`	Помилка значення – породжується якщо інструкція отримує аргумент правильного типу, але некоректного значення.

Створення екземпляру виключення

Щоб у програмі проаналізувати виключення детальніше, у блоці **`except`** вказують не лише тип виключення, але і створюють екземпляр цього виключення, що містить детальну інформацію про виключення. Для цього, у блоці **`except`** після типу виключення вказують ім'я екземпляру виключення

```
except exception as e:
```

тут e – ім'я екземпляру виключення.

Як було зауважено раніше, типи виключень це класи. Тоді екземпляри виключень це об'єкти, що належать до відповідних класів. Оскільки передбачається, що наразі читач ще не знайомий з об'єктно-орієнтованим програмуванням, то не будемо перевантажувати читача правилами роботи з екземплярами виключень, як з об'єктами класів. Єдине що важливо буде для нас, так це те, що екземпляр виключення можна вивести на екран з допомогою інструкції `print`.

```
try:
    print(1 / 0)
except ZeroDivisionError as e:    # Створюємо екземпляр e
    print(e)    # Виводимо екземпляр виключення e на екран
```

Результатом виконання цього блоку коду буде виведення на екран повідомлення

```
division by zero
```

Перевірка твердження

Щоб перевірити стан програми (наприклад для того, щоб переконатися, що подальше її виконання має сенс) використовують інструкцію **assert**. Вона має такий синтаксис

```
assert condition, message
```

де `condition` – умова (логічний вираз), `message` – необов'язковий параметр, що є даними, що передаються на обробку при виникненні виключної ситуації.

Інструкція **assert** генерує виключення типу `AssertionError`, якщо логічний вираз `condition` є хибним. Наприклад,

```
x = 0
try:
    assert x != 0
except AssertionError:
    print("Ділити на 0 не можна")
else:
    print(1 / x)
```

Ініціювання виключення

Програміст може самостійно ініціювати виключення будь-якого типу. Основне призначення цього механізму – це застосування механізмів обробки виключних ситуацій для створених користувацьких виключень.

Синтаксис ініціювання виключення такий

```
raise exception
```

тут `exception` – тип виключення (або екземпляр класу виключення).

При виконанні цієї інструкції Python діє аналогічно тому, як він діє при штатному виникненні виключення цього типу. Наприклад, у наведеному нижче блоці коду, програма ініціює виключення `ZeroDivisionError`, якщо змінна `x` (тобто другий операнд у операції ділення) дорівнює нулю.

```
x = 0
try:
    if x == 0:
        raise ZeroDivisionError     # Ініціювання виключення
except ZeroDivisionError: # Обробка виключення ділення на 0
    print("Ділення на 0")
else:          # Якщо виключення не відбулося виводимо 1 / x
```

```
    print(1 / x)
```

Менеджер контексту

Менеджери контексту дозволяють захопити та звільняти ресурси програми строго за необхідності, не залежно від того, що відбулося з програмою у ході її виконання.

Менеджер контексту реалізується за допомогою оператора **with**.

```
with expression as target:
    do_something
```

тут `expression` операція, що захоплює ресурси (наприклад відкриває файл, блокує ресурси тощо), `target` – об'єкт, що створюється в результаті операції `expression`.

Розглянемо правило за яким використовується менеджер контексту: нехай маємо дві пов'язаних операції, що мають бути виконані у парі (наприклад, відкриття і закриття файлу) і є код, що потрібно розмітити між ними (наприклад, обробка файлу)

```
захопити_ресурси f
обробити_дані f
звільнити_ресурси f
```

Тоді цей блок можна замінити оператором менеджера контексту

```
with захопити_ресурси as f:
    обробити_дані f
```

Отже, така операція з застосуванням менеджера контексту еквівалентна такому блоку з використанням оператора **try**:

```
захопити_ресурси f
try:
    обробити_дані f
finally:
    звільнити_ресурси f
```

і фактично є синтаксичною конструкцією, що дозволяє скоротити код програми і гарантує безпечне використання даних.

Задачі для самостійної роботи

7.1. Розв'язати задачу 3.59 для пунктів, де є обмеження на змінну x. Оформити перевірку вхідних даних за допомогою оператора **assert**.

7.2. Розв'язати квадратне рівняння $ax^2 + bx + c = 0$. Оформити перевірку вхідних даних (що рівняння є квадратним і має розв'язок на множині дійсних чисел) за допомогою оператора **assert**.

7.3. Знайти площу трикутника за трьома сторонами a, b, c. Оформити перевірку вхідних даних (що трикутник з такими сторонами a, b, c існує) за допомогою оператора **assert**.

7.4. Використовуючи операцію `index` (для списків), знайдіть позиції у заданому списку всіх елементів заданої множини.

7.5. Задано рядок, серед елементів якого містяться цифри. Використовуючи для всіх символів рядка функцію `int` перетворення символа у число, обчислити суму цифр заданого рядка.

7.6. Дано список чисел. Не використовуючи функцію `len` визначення кількості елементів у цьому списку та не використовуючи цикл по колекції, визначити:

a) кількість елементів у списку;
b) суму елементів у списку;
c) значення найбільшого відношення (частки) серед елементів.

7.7. Розв'язати задачу 6.12 використовуючи для словника лише операцію індексування за ключем (тобто операцію взяття елемента за ключем з допомогою оператора квадратні дужки).

7.8. До програми з клавіатури надходить послідовність цифр. Послідовність задається доти, щоки користувач не введе слово «досить». Слід зауважити, що користувач не є дисциплінованим і може заміть цифр вводити будь-що. Якщо користувач вводить з клавіатури число більше за 9, то програма ініціює виключення `RuntimeError`. Якщо користувач вводить число менше за 0, то програма ініціює виключення `TypeError`. Якщо користувач вводить дійсне значення з діапазону від 0 до 9, то програма ініціює виключення `ValueError`. Підрахувати кількість виключень кожного типу, що виникають у програмі.

§8 ПІДПРОГРАМИ

Підпрограми

Підпрограма – це логічно незалежний фрагмент коду, оформлений спеціальним чином за правилами мови програмування, до якого можна (багаторазово) звернутися з будь-якого місця програми.

Підпрограми дозволяють значно скоротити об'єм коду та спросити його розуміння.

Звернення до підпрограми називають **викликом підпрограми**.

Підпрограми бувають **іменованими** та **анонімними**

У загальному випадку у програмуванні підпрограми поділяють на **процедури** та **функції**.

Процедура – це фрагмент коду (послідовність інструкцій), призначений для розв'язання певної задачі.

Функція – процедура, що повертає у місце виклику певне значення, що є результатом її виконання.

У Python усі підпрограми є функціями.

З багатьма функціями ми вже зустрічалися (тобто використовували) при написанні програм раніше, наприклад

```
len(l)
max(l)
```

Це були вбудовані функції. Давайте познайомимося з тим, як створювати власні функції у Python, правилами виклику функцій та іншими особливостями роботи з функціями.

Опис функції

Перш ніж використовувати функцію її необхідно створити або, кажучи термінами програмування, **описати**.

Для опису функції у Python необхідно:

- задати ім'я функції (для іменованих функцій);
- описати аргументи функції (не обов'язково);
- описати програмний код функції;
- описати повернення результату (не обов'язково).

Отже, як видно з вищенаведеного, єдиною обов'язковою дією для опису функції у Python є опис програмного коду функції. Розглянемо синтаксис іменованих функцій (з анонімними функціями познайомимося пізніше)

```
def function_name(args):
    function_body
```

тут **def** – ключове слово, що вказує на початок опису функції, `function_name` – ім'я функції, що задається програмістом. `args` – послідовність аргументів (параметрів) функції, `function_body` – тіло функції, тобто програмний код функції.

Якщо опис функції не містить жодного аргументу, то `args` опускають. При цьому круглі дужки **не опускаються**:

```
def function_name():
    function_body
```

Для повернення результату використовується оператор

```
return result
```

де `result` – результат виконання функції. Інструкція **return** завершує виконання функції, навіть якщо за нею слідують інші інструкції. Тому її часто використовують у якості оператора примусового завершення функції, наприклад разом з умовним оператором або в циклах.

Зауважимо, що функція у Python може повертати будь-який об'єкт – число, рядок, список, кортеж, тощо. Якщо тіло функції не містить інструкції **return** або під час виклику функції інтерпретатор не натрапляє на неї, то функція повертає значення **None** ("ніщо").

Опишемо для прикладу функцію, що визначає більше з двох чисел:

```
def max2(a, b):
    if a > b:
        return a
    else:
        return b
```

Отже, ім'я цієї функції `max2`. Вона має два аргументи `a`, `b` та повертає значення `a`, якщо `a > b` і значення `b` у іншому випадку.

Тепер, після опису, функцію `max2` можна викликати:

```
>>> max2(2, 3)
3
```

```
>>> m = max2(444, 555)
>>> print(m)
555
```

Результат виклику функції можна використовувати як аргументи для виклику функції. Наприклад, описану вище функцію можна використовувати для знаходження найбільшого серед трьох чисел

```
>>> max2(11, max2(12, 13))
13
```

Повернення кількох значень функцією

Як було сказано раніше, за допомогою інструкції **return** можна повертати будь-який об'єкт Python, розпакований кортеж у тому ж числі. Цей механізм використовується для реалізації випадку повернення кількох значень однією функцією. Для цього необхідно у інструкції **return** через кому перерахувати значення, які повертає функція.

```
def func(...):
    ...
    return result1, result2, ..., resultN
```

Фактично така функція повертає кортеж

```
(result1, result2, ..., resultN)
```

При цьому його можна зразу розпакувати у послідовність змінних (кількість яких повинна дорівнювати довжині кортежу)

```
re1, res2, ..., resN = func(...)
```

Приклад 8.1. Визначити символ, що входить до рядка найбільшу кількість разів.

Розв'язок. Розіб'ємо задачу на дві частини:

1. Побудова на базі рядка словника, у якому ключами будуть символи, а значеннями кількість входжень відповідного символа до рядка.
2. Пошук у словнику ключа якому відповідає найбільше значення.

Оформимо кожну з частин у вигляді функції.

Перша функція `construct_dict` буде у якості параметра приймати рядок `S`, і повертати словник `char_dict` у якості результату.

```
def construct_dict(S):  # S - формальний параметр (рядок)
    char_dict = {}      # char_dict - словник символів
    for c in S:
        if c not in char_dict:
            char_dict[c] = S.count(c)
    return char_dict
```

Можемо перевірити роботу функції, викликавши її для деякого рядку, наприклад

```
>>> D = construct_dict("молоко")
>>> print(D)
```

Результат буде таким

```
{'л': 1, 'о': 3, 'к': 1, 'м': 1}
```

Друга функція `find_most_popular_char` буде приймати побудований словник і повертати пару значень – найпопулярніший символ і кількість його входжень

```
def find_most_popular_char(D):
    max_char = ''  # найпопулярніший символ
    max_num = 0    # кількість його входжень
    for c, num in D.items():
        if num > max_num:
            max_char = c
            max_num = num
    return max_char, max_num # функція повертає пару
                             # значень max_char, max_num
```

Для перевірити коректності роботи функції, здійснимо її виклик для раніше побудованого словника

```
>>> D = construct_dict("молоко")
>>> C, N = find_most_popular_char(D)
>>> print(C, N)
о 3
```

Перевіривши правильність роботи обох функцій, можемо написати головну програму. Отже, остаточно програма, що виконує поставлене у задачі завдання буде мати вигляд

```
def construct_dict(S):
    ...

def find_most_popular_char(D):
    ...

S = input("Задайте рядок ")
D = construct_dict(S)
C, N = find_most_popular_char(D)
print("символ '%s' входить у рядок %d разів" % (C, N))
```

Тут символами `...` умовно позначаються раніше наведені тіла функцій.

Аргументи функції

Аргументи функції (їх також називають **параметрами**), це вхідні параметри функції, які передаються функції на обробку. Від них залежить не лише поведінка функції, а й результат її виконання.

Аргументи функції розділяють на формальні параметри та фактичні аргументи. Формальні параметри задаються під час опису функції. Формальні параметри не мають конкретних значень – вони лише використовуються для опису шаблону функції. Наприклад, у вищенаведеному описі функції

```
def max2(a, b):
    ...
```

`a, b` є формальними параметрам.

Фактичні аргументи вказуються **під час виклику функції**. Фактичні аргументи мають конкретні значення (літерал, значення виразу, об'єкт). Наприклад, під час виклику функції

```
>>> max2(2, 3)
3
```

`2` і `3` є фактичними аргументами.

Передача параметрів у функцію

Фактично будь-яка функція є шаблоном коду, що залежить від формальних параметрів. Щоб виконати функцію її потрібно викликати з фактичними аргументами. При цьому відбувається **передача параметрів у функцію**.

Розберемо передачу параметрів у функцію на прикладі. Нехай описано функцію

```
def func(param1, param2, param3):
    ...
```

що має три формальних параметри `param1, param2, param3`.

Під час виклику функції з фактичними аргументами `arg1, arg2, arg3`

```
func(arg1, arg2, arg3)
```

фактичні параметри підставляються в шаблон функції замість формальних, тобто відбувається таке присвоєння

```
param1 = arg1
param2 = arg2
param3 = arg3
```

після чого відбувається виконання тіла функції. Це присвоєння і буде передачею параметрів у функцію.

При цьому потрібно пам'ятати, що операція присвоювання копіює не значення змінних (або об’єктів), а їхні адреси (посилання) у пам’яті. Тобто, після вищенаведеного присвоєння Змінні `param1` та `arg1` будуть посилатися на один і той же об'єкт в пам'яті. Аналогічна ситуація буде і з іншими парами змінних `param2` та `arg2` і `param3` та `arg3`. Використовуючи термінологію програмування, кажуть, що передача параметрів у Python відбувається **за посиланням**, а не **за значенням** (як наприклад, у мовах програмування Pascal або C\C++).

Передача параметрів за посиланням вимагає від програміста більше обережності при написанні функцій, Наприклад, при зміні значень формальних параметрів в тілі функції, що належать до незмінюваних об'єктів, фактичні аргументи не зміняться, а от зміни над формальними аргументами, що є змінюваними відобразяться на зміні фактичних параметрів.

Щоб краще зрозуміти вищенаведене, наведемо приклади.

Опишемо функцію, що збільшує значення аргументу на 1.

```
def increment_Int(a):
    a += 1      # збільшуємо значення змінної a на 1
```

Будемо викликати функцію для змінної `myInt` зі значенням 1. Після чого виведемо значення змінної `myInt` на екран

```
myInt = 1
increment_Int(myInt)
print(myInt)
```

Результатом буде

```
1
```

Спробуємо розібратися чому значення змінної `myInt` не змінилося, якщо у тілі функції ми збільшили `a`, а значить і `myInt` на 1.

В момент виклику функції `increment_Int` обидві змінні – і формальний параметр `a` і фактичний аргумент `myInt` посилаються на один об'єкт в пам'яті, що належить до цілого типу. Після виконання інструкції `a += 1` (оскільки `a` належить до `immutable` типу), формальний параметр буде посилатися вже на інший об'єкт цілого типу зі значенням **2**, а фактичний аргумент залишиться незмінним.

Розглянемо інший приклад, у якому фактичний аргумент буде належати до змінюваного типу. Опишемо функцію, що додає до списку число **1**.

```
def increment_list(a):
    a.append(1)    # додаємо до списку a число 1

myList = [1]
increment_list(myList)
print(myList)
```

результатом виконання вищенаведеного коду буде

```
[1, 1]
```

Аналогічно попередньому прикладу, в момент виклику функції `increment_list` обидві змінні – і формальний параметр `a` і фактичний аргумент `myList` посилаються на один і той же список. Після виконання інструкції `a.append(1)` (оскільки `a` належить до змінюваного типу), формальний параметр `a`, а разом з ним і фактичний аргумент `myList` доповняться елементом **1**.

Типові значення аргументів

Для формальних параметрів функцій часто задають **типові значення** (eng: "by default", рус: "по умолчанию"). Типові значення функції, це такі значення, які будуть використовуватися функцією, якщо аргумент функції явно не заданий під час виклику цієї функції.

Для того, щоб задати типове значення формальному параметру, його задають в описі функції через оператор присвоєння, наприклад:

```
def func(param = "Типове значення"):
    print(param)
```

Здійснимо виклик описаної функцію з заданим фактичним аргументом `"Моє значення"` та з використанням типового значення.

```
>>> func("Моє значення")
Моє значення
>>> func()
Типове значення
```

Як ми знаємо, функція може мати довільну кількість аргументів. Частина з цих аргументів може мати типові значення, а інша частина – ні. Існує важливе правило для використання типових значень: У описі функції, у переліку формальних параметрів спочатку задаються параметри, що не мають типових значень, а аргументи, що мають типові значення задаються у описі функції в кінці.

```
def func(p1, p2, p3 = u, p4 = v):
    ...
```

Під час виклику функції, типові значення задаються відповідно до порядку у списку аргументів. Наприклад, якщо для вищенаведеної функції виклик буде мати вигляд

```
func(a1, a2, a)
```

то це означатиме, що для параметру `p4` використовується типове значення `v`, а для `p3` задане значення `a`.

Приклад 8.2. За допомогою розкладу функції e^x в ряд Тейлора

$$y(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

обчислити з точністю $\varepsilon > 0$ її значення для заданого значення x.

Розв'язок. Поставлена задача була розв'язана у прикладі 3.26. Оформимо код наближеного обчислення у вигляді функції.

Як було встановлено раніше, функція e^x визначається як границя послідовності S_n, що задана системою рекурентних співвідношень

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

При цьому під наближеним (з точністю ε) значенням границі послідовності S_n будемо розуміти такий член S_N , що виконується співвідношення

$$|S_N - S_{N-1}| = |a_N| < \varepsilon$$

Отже, опишемо функцію, що буде мати два формальних параметри `x`, `eps`, що відповідають x і ε, причому формальний параметр `eps` буде мати типове значення `0.0001`.

```
# Описуємо функцію exp
def exp(x, eps = 0.0001):
    S = a = 1
    n = 0
    while abs(a) >= eps:
        n += 1
        a *= x / float(n)
        S += a
    return S

# головна програма
x = float(input("x = "))
y = exp(x) # використовуємо типове значення параметра eps
print ("exp(%f) = %f" % (x, y))
```

Позиційні та ключові параметри

Аргументи функцій, розглянуті раніше, належать до **позиційних параметрів**, тому що співставлення між фактичними аргументами та формальними параметрами здійснюється за позицією у списку параметрів.

Проте, таке співставлення можна здійснювати не за позицією, а за відповідним іменем формального параметра. Наприклад, для функції

```
def func(param1, param2, param3):
    ...
```

виклик може мати вигляд

```
y = func(param1 = a1, param2 = a2, param3 = a3)
```

Такі фактичні аргументи (`a1, a2, a3`) називаються **ключовими**, оскільки передача аргументів функції здійснюється парами – ключ (ім'я формального параметра)-значення (ім'я фактичного аргументу).

Основна перевага ключових параметрів у тому, що їхній порядок розміщення під час виклику функції не має значення. Наприклад, виклик вищенаведеної функції може виглядати і таким чином

```
y = func(param2 = a2, param3 = a3, param1 = a1)
```

Функції з довільною кількістю параметрів

У Python можна описувати функції, що приймають довільну кількість параметрів, причому, як позиційних, так і ключових.

Для довільної кількості позиційних параметрів використовують оператор `*` перед іменем аргументу.

```
def func(*args):
    ...
```

Під час виклику такої функції аргументи передаються звичайним чином – записуються через кому:

```
x = func()            # виклик функції без аргументів
func(x, y)            # виклик функції з двома аргументами
func(a1, a2, a3, a4) # виклик функції з чотирма аргументами
```

Параметр `args` (без `*`) при такому описі функції є кортежом. Переконатися у цьому можна на такому простому прикладі

```
>>> def func(*args):
        print(args)

>>> func(1, 2.0, "string")
(1, 2.0, 'string')
```

Приклад 8.3. Опишемо функцію, що знаходить суму елементів заданої послідовності.

Розв'язок. Оскільки не відомо, скільки аргументів буде мати функція, опишемо її як таку, що приймає довільну кількість позиційних аргументів.

```
def suma(*elements):
    res = 0
    # Оскільки elements – кортеж скористаємося циклом for
```

```
    for el in elements:
        res += el
    return res

# Головна програма полягає у виклиці функції suma.
print(suma(1, 2, 3, 4, 5))
```

Для довільної кількості ключових параметрів використовують оператор `**` перед іменем аргументу.

```
def func(**kwargs):
    ...
```

Параметр `kwargs` при цьому є словником у якому формальні параметри є ключами, а фактичні – значеннями.

```
>>> def func1(**kwargs):
        print(kwargs)

>>> func1(x = 1, y = 2, z = 3)
{'x': 1, 'z': 3, 'y': 2}
```

Зауважимо, що в Python можна описувати функції, що приймають довільну кількість позиційних і ключових параметрів. Стандартний опис тоді такої функції матиме вигляд

```
def func(*args, **kwargs):
    ...
```

Локальні та глобальні змінні

При описі функцій у тілі функції ми часто використовували змінні, що не були формальними параметрами. Наприклад, у описі функції

```
def suma(*elements):
    res = 0
    for el in elements:
        res += el
    return res
```

такими змінними є `res` та `el`.

Такі змінні, які визначені в тілі функції називаються **локальними** і є доступними лише в тілі функції у якій визначені. Щоб переконатися у цьому опишемо функцію, що має локальну змінну та спробуємо скористатися цією змінною поза межами тіла функції

```
def func():
    s = 1  # s - локальна змінна

func()
print(s)
```

Результатом виконання вищенаведеного коду буде повідомлення про помилку, у якому буде сказано, що змінна `s` не є визначеною:

```
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    s
NameError: name 's' is not defined
```

Змінні, що є досяжними поза функцією, називаються **глобальними**.

```
x = 1          # x - глобальна змінна

def func():
    print(x) # Виводимо на екран глобальну змінну x

func()         # Викликаємо вищеописану функцію
```

Результатом виконання вищенаведеного коду буде

```
1
```

Не розуміння різниці між локальними і глобальними змінними часто призводить до плутанини і, відповідно, до помилок у коді.

Щоб краще засвоїти різницю між локальними та глобальними змінними, розглянемо кілька прикладів.

```
x = 1          # x - глобальна змінна зі значенням 1

def func():
    print(x) # Виводимо на екран глобальну змінну x
```

```
func()          # Викликаємо вищеописану функцію,
                # що виводить на екран x
print(x)        # Виводимо на екран глобальну змінну x
```

Результатом виконання цього коду буде

```
1
1
```

Виконаємо вищенаведену програму, модифікувавши функцію `func`.

```
x = 1           # x - глобальна змінна зі значенням 1

def func():
    x = 222     # Змінимо значення x
    print(x)    # Виводимо на екран глобальну змінну x

func()          # Викликаємо вищеописану функцію,
                # що виводить на екран x
print(x)        # Виводимо на екран глобальну змінну x
```

Результатом виконання цього коду буде

```
222
1
```

Такий результат пояснюється тим, що у першому випадку, інтерпретатор, не знайшовши у функції явного опису змінної `x`, зрозумів, що треба використовувати глобальну змінну `x`. У другому випадку, операцію присвоєння

```
    x = 222     # Змінимо значення x
```

інтерпретатор сприйняв не як модифікацію глобальної змінної, а як створення нової змінної. Глобальна ж змінна `x` при цьому лишилася незмінною.

Виникає питання: що ж робити, якщо в тілі функції необхідно модифікувати глобальну змінну?

Модифікація глобальних змінних у тілі функції є небезпечним та може призвести до неочікуваних результатів під час виконання програми! Застосування такого підходу повинно бути обґрунтованим. У загальному випадку, для обміну даними між функцією і зовнішнім світом рекомендується використовувати механізм параметрів.

Для того, щоб у тілі функції явно вказати, які зі змінних є глобальними, використовують ключове слово **global.** Після цього слова вказують які змінні (через кому, якщо їх кілька) необхідно трактувати як глобальні.

```
x = 1          # x – глобальна змінна зі значенням 1

def func():
    global x # Вказуємо, що використовується глобальна x
    x = 222  # Змінимо значення x (глобальної)
    print(x) # Виводимо на екран глобальну змінну x

func()         # Викликаємо вищеописану функцію,
               # що виводить на екран x
print(x)       # Виводимо на екран глобальну змінну x
```

Результатом виконання цього коду буде

```
222
222
```

Функціональний тип

Опис функції призводить до створення об'єкту функціонального типу. Ім'я функції у цьому випадку є нічим іншим як змінною (посиланням на цей об'єкт), через яку можна здійснювати операцію виклику цієї функції.

Отже з точки зору Python не існує жодної концептуальної різниці між створенням функції і створенням звичайної змінної (об’єкту).

Як відомо зі змінними можна здійснювати операції, які визначаються типом літерала, до якого належить значення змінної. Наприклад, над числами можна проводити арифметичні операції (додавання, віднімання, множення і т.д.), над рядками і списками операції конкатенації, по-елементного проходження тощо. Єдина операція, яку можна здійснювати над функцією – викликати її.

Щоб краще зрозуміти вищезазначене, розглянемо приклад. Опишемо таку функцію

```
# Оголошуємо функціональний об'єкт (функцію) з іменем func
def func(p):
    print("Функція у точці {}".format(p))
```

Будемо вважати, що опис здійснено в окремому файлі, а приклад роботи з цією функцією здійснюється у інтерактивному режимі.

```
>>> func(5)       # Здійснимо виклик func для аргумента 5
Функція у точці 5
>>> func          # Виведемо на екран func
<function func at 0x01A484B0>
>>> func1 = func # func1 і func посилаються на один об'єкт
>>> func1         # Виведемо на екран func1
<function func at 0x01A484B0>
>>> func1(5)
Функція у точці 5
```

Отже, оскільки функції нічим не відрізняються від інших об’єктів Python, то функції можуть передаватися у якості аргумента іншим функціям. У тілі ж функції до такого параметру можна застосовувати операцію виклику

```
# Функція func має формальний параметр f який є функцією
# тобто посиланням на функціональний об'єкт
def func(..., f): # параметр f – фукнція
    ...
    x = f(...)     # Здійснюється виклик функції f
                   # що є аргументом функції func
    ...
```

Приклад 8.4. Для заданої функції потрібно побудувати таблицю її значень.

Розв'язок. Згідно з умовою задачі нам необхідно побудувати, наприклад, для функції $f = x^2$ для точок $\{0, 1, 2, 3, 4\}$ таку таблицю

x	0	1	2	3	4
$f = x^2$	0	1	4	9	16

Звичайно цю задачу можна реалізувати для конкретної функції. Проте зробимо універсальну функцію, що буде будувати таблицю для довільної заданої функції.

Отже вхідними параметрами у нашу функцію буде список точок і функція, для якої буде будуватися таблиця.

Таблицю оформимо у вигляді словника, у якій ключами будуть точки, а значеннями – значення функції у відповідній точці.

```
import math

def table(l, f):     # l – список точок, f – функція
    d = {}           # Словник d – таблиця значень функції
    for x in l:      # Проходимо по всіх x
        d[x] = f(x) # Додаємо пару (x, f(x)) у словник
    return d         # Повертаємо таблицю (словник)
```

```
# Створюємо список точок, що є розбиттям відрізка
# [0, pi] на 5 частин
l = [x * math.pi / 10 for x in range(6)]
# Будуємо таблицю значень функції sin для заданого списку
D = table(l, math.sin)

# Виводимо результат на екран у вигляді sin(0) = 0
for x in l:
    print("sin(%f) = %f" % (x, D[x]))
```

Результат виконання цієї програми

```
sin(0.000000) = 0.000000
sin(0.314159) = 0.309017
sin(0.628319) = 0.587785
sin(0.942478) = 0.809017
sin(1.256637) = 0.951057
sin(1.570796) = 1.000000
```

Як і звичайну змінну, функцію можна створювати у тілі іншої функцій. Більше того, створену функцію можна повернути у ролі результату за допомогою оператора **return** та використовувати отриману функцію у головній програмі.

Приклад 8.5. Обчислимо значення виразу

$$1 + x^2 + \cdots + x^n$$

для заданих дійсного числа x і натурального n.

Розв'язок. Опишемо функцію `pow_nat()`, що будує функцію, яка обчислює степінь порядку n для заданого аргументу x.

```
def pow_nat(n):

    def tmp_pow(x):
        res = 1
        for i in range(n):
            res *= x
        return res

    return tmp_pow

n = int(input("n= "))
x = float(input("x= "))
```

```
res = 0
for k in range(n + 1):
    res+= pow_nat(k)(x)
print(res)
```

Звернемо вашу увагу на те, що функція `pow_nat`, з одним аргументом повертає у якості результату функцію $f(x) = x^k$. Тобто інструкцію `pow_nat(k)` можна розглядати, як ім'я такої функції. Отже, щоб обчислити значення x^k, для заданого x використовуємо інструкцію `pow_nat(k)(x)`.

Анонімні функції

Анонімні функції (або лямбда-функції), це особливі функції, що описуються у місці їхнього виклику і не отримують імені (ідентифікатора) для доступу до них.

У Python анонімні функції можуть складатися лише з одного виразу.

Створюються анонімні функції з допомогою лямбда-виразу, що має такий синтаксис

```
lambda args: result
```

тут **`lambda`** – ключове слово, що вказує на опис анонімної функції, `args` – послідовність аргументів (параметрів) функції. `result` – результат, що повертає lambda-функція.

Як і під час створення звичайної функції, під час створення анонімної функції створюється об'єкт функціонального типу. Проте lambda-функції не отримують ідентифікатора для їхнього виклику. Основне призначення лямбда-функцій – використовуватися у ролі аргументу іншої функції.

Суттєвою перевагою лямбда-функцій над звичайними функціями є те, що вони виконуються значно швидше.

```
>>> (lambda x, y: x + y)(2, 3) # одночасне створення і виклик
5
>>> f = lambda x, y: x + y #f – посилання на лямбда-функцію
>>> f(2, 3)          # Виклик лямбда-функції через посилання f
5
```

Розглянемо застосування лямбда-функції до прикладу 8.4, у якому будувалася таблиця значень для заданої функції. Опис функції `table` опишемо схематично, оскільки він нічим не відрізняється від того, що наведений у прикладі 8.4**.** Побудуємо таблицю значень для функції $f = x^2$, для чого скористаємося відповідною лямбда-функцією. Отже програма буде мати вигляд

```
def table(l, f):
    ...

# l – розбиття відрізка [0, 1] на 10 частин
l = [x / 10.0 for x in range(11)]
# Будуємо таблицю значень функції x**2
D = table(l, lambda x: x**2)# у якості другого аргументу
                            # використовується лямбда-вираз
# Виводимо результат на екран у вигляді f(0.0) = 0.0
for x in l:
    print("f(%f) = %f" % (x, D[x]))
```

Рекурсія

Рекурсія – спосіб визначення об'єкту (або методу) попереднім описом одного чи кількох його базових випадків, з наступним описом на їхній основі загального правила побудови об'єкту.

З рекурсією ми часто зустрічаємося у оточуючому житті: рекурсивні зображення, структура рослин та кристалів, рекурсивні розповіді, вірші або жарти.

Рекурсивна функція – метод визначення функції, при якому функція прямо або неявно викликає сама себе.

Розглянемо вищенаведене означення на прикладі.

Як ми знаємо послідовність чисел Фібоначчі визначається рекурентним співвідношенням другого порядку

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

Отже, якщо покласти, що функція $F(n)$ визначає n-те число послідовності Фібоначчі, то з вищенаведеного рекурентного співвідношення отримаємо, що послідовність Фібоначчі може бути визначена рекурсивною функцією:

$$\begin{cases} F(0) = 1,\ F(1) = 1, \\ F(n) = F(n-1) + F(n-2),\ n \geq 2. \end{cases}$$

Початкові значення $F(0) = 1, F(1) = 1,$ дуже важливі при визначенні рекурсивної функції. Якби їх не було, то рекурсія б стала нескінченною! Тому, описуючи рекурсивну функцію завжди треба переконуватися, що для всіх допустимих значень аргументів виклик рекурсивної функції завершиться, тобто що рекурсія буде скінченною.

Можна звернути увагу, на те, що на відміну від обчислення елементів послідовності заданої рекурентним співвідношенням, де обчислення відбувається він тривіального (початкового) елементу до шуканого, рекурсивна функція починає обчислення від шуканого.

Кількість вкладених викликів функції називається **глибиною рекурсії**. Наприклад, глибина рекурсії при знаходженні $F(5)$ буде 4.

Опишемо стандартний алгоритм опису рекурсивної функції. Структурно рекурсивна функція на верхньому рівні завжди є розгалуженням, що складається з двох або більше альтернатив, з яких

- принаймні одна є термінальною (припинення рекурсії);
- принаймні одна є рекурсивною (тобто здійснює виклик себе з іншими аргументами).

```
def recursive_func(args):
    if terminal_case:        # Термінальна гілка
        ...
        return trivial_value # Тривіальне значення
    else:                    # Рекурсивна гілка
        ...
        # Виклик функції з іншими аргументами
        return expr(recursive_func(new_args))
```

Для прикладу опишемо рекурсивну підпрограму для обчислення чисел Фібоначчі. У програмі цю функцію будемо позначати `Fib(n)`.

```
def Fib(n):
    if n == 0 or n == 1:                # Термінальна гілка
        return 1                        # Тривіальне значення
    else:                               # Рекурсивна гілка
        return Fib(n - 1) + Fib(n - 2) # Рекурсивний виклик
# Виклик рекурсивної функції
n = int(input("n = "))
print("Fib(%d) = %d" % (n, Fib(n)))
```

Рекурсія використовується, коли можна виділити **самоподібність** задачі. Розглянемо інший класичний приклад, який використовується у навчальній літературі для пояснення рекурсії.

Приклад 8.6. Описати рекурсивну функцію для знаходження факторіала натурального числа.

Розв'язок. Очевидно, що функцію $F(n) = n!$ можна задати у такому рекурсивному вигляді

$$\begin{cases} F(0) = 1, \\ F(n) = nF(n-1),\ n \geq 1. \end{cases}$$

Тоді програма, разом з рекурсивною функцією (у програмі будемо позначати її `Fact(n)`) буде мати такий вигляд

```
def Fact(n):
    if n == 0 :                 # Термінальна гілка
        return 1                # Тривіальне значення
    else:                       # Рекурсивна гілка
        return n * Fact(n - 1)  # Рекурсивний виклик
# Виклик рекурсивної функції
n = int(input("n = "))
print("%d! = %d" % (n, Fact(n)))
```

Приклад 8.7. Описати рекурсивну функцію перевірки рядка на симетричність.

Розв'язок. Спочатку опишемо термінальні випадки. Для цього проаналізуємо найпростіші рядки. Очевидно, що порожній рядок або рядок, що складається з одного символа (наприклад, `""`, `"a"`) є симетричним.

Тепер побудуємо рекурсивний виклик. Очевидно, що якщо перший і останній символи рядка є однаковими і рядок без них є симетричним, то вихідний рядок є симетричним.

```
# Рекурсивна функція перевірки рядка s на симетричність
def is_symetric(s):
    if len(s) <= 1:     # якщо рядок s порожній або
                        # складається з одного символа
        return True     # то він симетричний
    else:
        # cond1 - умова, що 1-й і останній символи однакові
        cond1 = s[0] == s[len(s)-1]
        # cond2 - умова, що рядок без першого і останнього
        # символа є симетричним - рекурсивний виклик
        cond2 = is_symetric(s[1 : len(s) - 1])
        # рядок симетричний, якщо обидві умови істині
        return cond1 and cond2
# Виклик рекурсивної функції для введеного з клавіатури рядка
S = input("S = ")
sym = is_symetric(S)

if sym:
    print("Заданий рядок є симетричним")
```

```
else:
    print("Заданий рядок не є симетричним")
```

Питання про використання рекурсії дуже суперечливе і неоднозначне. З одного боку, рекурсивна форма як правило значно простіша і наглядніша. З іншого боку, рекомендується уникати рекурсивних алгоритмів, що можуть приводити до занадто великої глибини рекурсії, особливо у випадках, коли такі алгоритми мають очевидну реалізацію у вигляді звичайного циклічного алгоритму. З огляду на це, вищенаведений рекурсивний алгоритм визначення факторіала є прикладом скоріше того, як не треба застосовувати рекурсію.

Теоретично будь-яку рекурсивну функцію можна замінити циклічним алгоритмом (можливо з застосуванням стеку).

Функції-генератори

Функція-генератор – спеціальний об'єкт Python, що (подібно до генератор-виразів) будує послідовність елементів деякої послідовності з по-елементним доступом до її членів.

Подібно до генератор-виразу, функція-генератор не будує всю послідовність одразу, а повертає її члени при кожному зверненні. Ця особливість дуже корисна при обробці великого масиву даних, оскільки не потрібно завантажувати весь масив, що економить час та оперативну пам'ять.

Опис генератор-функцій має такий же синтаксис, як і звичайна функція, за виключенням того, що для повернення результату (тобто поточного члена послідовності) замість оператора

```
return result
```

використовується оператор

```
yield result
```

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

Приклад 8.8. Опишемо генератор-функцію побудови чисел Фібоначчі. Використаємо її для виведення перших N чисел Фібоначчі.

Розв'язок. Опишемо спочатку генератор-функцію, що будує перші N чисел послідовності Фібоначчі та використаємо її.

```
def Fib(n):
    F1 = F2 = 1
    yield F2
    yield F1
    for i in range(3, n+1):
        F2, F1 = F1, F1 + F2
        yield F1

N = int(input("N = "))
# Створюємо об'єкт генератор-функцію
generator = Fib(N)

for a in generator:
    print(a, end=" ")
```

Оскільки генератор-функція не будує одразу всієї послідовності, то можна описувати генератор-функції для визначення як завгодно великих членів необмеженої послідовності. Перепишемо вищенаведений приклад побудови необмеженої послідовності за допомогою генератор-функції

```
def Fib():
    F1 = F2 = 1
    yield F2
    yield F1
    while True:
        F2, F1 = F1, F1 + F2
        yield F1

N = int(input("N = "))
i = 1
for a in Fib():  # Використання генератор-функції Fib
    print(a, end=" ")
    if i >= N:
        break
    i += 1
```

У цьому прикладі ми використовували генератор-функцію у циклі **for**, який відповідав за ітерацію елементів послідовності. Проте, генератор-функції можна використовувати і поза межами циклу по колекції. Для цього використовується функція `next`, яка повертає наступне значення генератор-функції:

```
next(generator)
```

тут `generator` – деякий об'єкт генератор-функції.

Змінимо вищенаведену програму з використанням функції `next`. Оскільки опис генератор-функції лишиться без змін, то у програмі опишемо його схематично

```
def Fib():
    ...

# створення об'єкту генератор-функції
generator  = Fib()

N = int(input("N = "))
for i in range(N):
    # Виводимо на екран наступне значення
    print(next(generator))
```

Для генератор-функцій, що будують обмежену послідовність (як у найпершому прикладі для послідовності Фібоначчі) функція `next` генерує виключення `StopIteration`, якщо відбувається спроба звернутися до неіснуючого елементу послідовності. Тому рекомендується оточувати виклик функції `next` блоком **`try…except`**.

```
def Fib(n):
    ...

# створення об'єкту генератор-функції
generator  = Fib(10)
while True:
    try:
        print(next(generator))
    except StopIteration:
        break
```

Декоратори для функцій

Означення

Як було вище зазначено, функції можна передавати у інші функції у ролі аргументів, а також функції можуть бути результатами роботи функцій. Цей факт використовується для створення спеціальних функцій, які називаються декораторами.

Декоратор – це функція, що дозволяє змінити поведінку функції (отриманої у якості аргументу) не змінюючи коду самої функції.

Як правило декоратори використовуються для того, щоб додати додаткові можливості функціям. Власне сенс застосування декоратора і полягає у тому, що вони можуть додавати однакову поведінку різним функціям. Наприклад, широко використовуються декоратори, що обчислюють час роботи функції, перевіряють коректність аргументів функцій, контролюють безпеку даних під час виконання функції тощо.

Щоб створити декоратор потрібно створити функцію, яка у якості аргументу буде отримувати функцію, модифікувати отриману функцію у тілі декоратора та повернути модифіковану функцію. Розглянемо такий приклад

```
def decorator(function):

    # Створюємо нову функцію
    def decorated_function():
        print("Код, що буде виконано до виклику функції")
        function()  # Виклик функції, що декорується
        print("Код, що буде виконано після виклику функції")

    # Повертаємо декоровану функцію
    return decorated_function
```

Виникає запитання: як скористатися створеним декоратором? Визначимо функцію

```
def seyHello():
    print("Усім вітання від функції seyHello!")
```

Задекоруємо її з допомогою декоратора `decorator`. Для цього необхідно змінній `seyHello` (що є посиланням на функцію у пам'яті) присвоїти результат виконання декоратора `decorator` над нею:

```
seyHello = decorator(seyHello)
```

Результатом виклику функції

```
seyHello()
```

буде

```
Код, що буде виконано до виклику функції
Усім вітання від функції seyHello!
Код, що буде виконано після виклику функції
```

Для спрощення розуміння коду, декорування функцій під час їхнього опису, можна здійснювати за допомогою оператора @. Наприклад, вищенаведені опис функції `seyHello` та її декорування

```
def seyHello():
    print("Усім вітання від функції seyHello!")
seyHello = decorator(seyHello)
```

можна замінити рівносильною їм синтаксичною конструкцією

```
@decorator
def seyHello():
    print("Усім вітання від функції seyHello!")
```

Загальний опис декораторів

У попередньому прикладі декоратор `decorator` міг застосовуватися лише до функцій, з порожнім списком аргументів. Загальний шаблон декоратора, що може застосовуватися до функцій з будь-якою кількістю як позиційних так і ключових аргументів, має такий вигляд:

```
def decorator(function):
    # Створюємо нову функцію
    def decorated_function(*args, **kwargs):
        # Код, що буде виконано до виклику функції
        ...
        res = function(*args, **kwargs)  # Виклик функції
        # Код, що буде виконано після виклику функції
        ...
        return res
    # Повертаємо декоровану функцію
    return decorated_function
```

Приклад 8.9. Опишемо декоратор, що вимірює час виконання функції. Застосуємо його для перевірки часу обчислення чисел Фібоначчі з використанням нерекурсивного і рекурсивного варіантів.

Розв'язок. Опишемо спочатку декоратор. Тут буде використовуватися функція `clock()` з бібліотеки `time`, що повертає поточний системний час.

Будемо зчитувати та запам'ятовувати час до початку виконання функції та після її виконання та виводити різницю на екран.

```
from time import clock # підключення функції clock
def benchmark(f):
    def _benchmark(*args, **kw):
        #вимірюємо час перед викликом функції
        current_time = clock()
        rez = f(*args, **kw)  #викликаємо f
        #обчилюємо різницю у часі
        dt = clock() - current_time
        print('Час виконання функції %1.5f сек' % dt)
        return rez
    return _benchmark
```

Опишемо функції `Fib1(n)` для нерекурсивного і `Fib2(n)` для рекурсивного варіантів обчислення `n`-го числа Фібоначчі, та задекоруємо їх за допомогою вищенаведеного декоратора `benchmark`. Оскільки декоратор діє на функцію, то для рекурсивного варіанту ми змушені використати допоміжну функцію `FibRecursive(n)`, яка буде реалізовувати рекурсію, а функція `Fib2` буде лише викликати функцію `FibRecursive(n)`.

```
@benchmark
def Fib1(n):
    F1 = F2 = 1
    for i in range(2, n + 1):
        F2, F1 = F1, F1 + F2
    return F1

def FibRecursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return FibRecursive(n-1) + FibRecursive(n-2)

@benchmark
def Fib2(n):
    return FibRecursive(n)
```

Виклик функцій нічим не відрізнятиметься від звичайного виклику недекорованих функцій:

```
N = 30
print(Fib1(N))
print(Fib2(N))
```

Проте, результатом роботи програми буде:

```
Час виконання функції 0.00001 сек
1346269
Час виконання функції 0.61396 сек
1346269
```

Вкладені декоратори

До функції може бути застосовано кілька декораторів одночасно. Їх вказують перед описом функції. Вказані декоратори застосовуються до функції послідовно, починаючи з найближчого до опису функції декоратора. Тому таке застосування декораторів називається їхнім вкладенням. Отже, якщо маємо код

```
@decorator1
@decorator2
def func():
    ...
```

то це означатиме, що до функції `func` спочатку застосовується декоратор `decorator2`, а вже потім до їхнього результату декоратор `decorator1`.

Задачі для самостійної роботи

8.1. Два простих числа називаються "близнюками", якщо вони відрізняються одне від одного на 2 (наприклад, числа 41 та 43). Скласти програму виведення на друк усіх пар "близнюків" із відрізка $[n,2n]$, де n – задане ціле число, яке більше за 2. У програмі описати функцію, що визначає чи є задане натуральне число простим.

8.2. Дано натуральне число n та послідовність натуральних чисел $a_1, a_2, \dots a_n$. Показати всі елементи послідовності, які є:

a) повними квадратами;

b) степенями п'ятірки;

c) простими числами.

Вказівка. Визначити відповідні функції для перевірки, чи є число повним квадратом, степенем п'ятірки, простим числом.

8.3. Дано натуральне число n. Для чисел від 1 до n визначити всі такі, які можна зобразити у вигляді суми двох повних квадратів. Описати функцію, яка

перевіряє, чи є число повним квадратом.

8.4. Щасливим називають таке шестизначне число, у якого сума перших трьох дорівнює сумі останніх трьох. Знайти всі щасливі числа. Описати функцію для визначення суми цифр тризначного числа.

8.5. Дано парне число $\mathrm{n} > 2$. Перевірити для нього гіпотезу Гольдбаха про те, що кожне парне число $\mathrm{n} > 2$ можна зобразити у вигляді суми двох простих чисел. Визначити функцію, яка перевіряє, чи є число простим.

8.6. Скласти програму обчислення величини

$$\frac{\sqrt[3]{a} - \sqrt[6]{a^2 + 1}}{1 + \sqrt[7]{3 + a}}$$

для заданого дійсного числа $\mathrm{a} > 0$. Визначити функції обчислення коренів $y = \sqrt[k]{x}$ із заданою точністю (див. задача 3.62).

8.7. Визначити підпрограми для наближених обчислень значень функцій, використовуючи їхні розклади в ряд Тейлора (див. задача 3.59).

8.8. Дано координати вершин трикутника й точки всередині його. Використовуючи функцію для обчислення площі трикутника через три його сторони, визначити відстань від даної точки до найближчої сторони трикутника.

Вказівка. Врахувати, що площа трикутника обчислюється також через основу й висоту.

8.9. Описати функцію, що обчислює наближене значення визначеного інтегралу від заданої функції на заданому проміжку. Необхідне наближене значення обчислити використовуючи суми Дарбу. Здійснити виклик описаної функції для:

a) стандартних функцій з модуля math;
b) для функцій побудованих у прикладі 8.7;
c) для функцій-поліномів, заданих за допомогою лямбда-функцій.

8.10. Порахувати кількість компонент вектору, які належать відрізку [a, b]. Описати функції для введення вектора та обчислення кількості компонент, а також лямбда-функцію, що перевіряє приналежність числа відрізку.

8.11. Визначити рекурсивні підпрограми зображення натурального числа:

a) у двійковій системі числення;
b) у вісімковій системі числення:
c) у системі числення з основою $b \geq 2$.

8.12. Визначити рекурсивну функцію обчислення $\mathrm{НСД}(n, m)$ натуральних чисел, яка ґрунтується на співвідношенні $\mathrm{НСД}(n, m) = \mathrm{НСД}(m, r)$, де r – залишок від ділення n на m.

8.13. Визначити рекурсивну функцію обчислення степеня дійсного числа з цілим показником x^n, згідно з формулою

$$x^n = \begin{cases} 1, & n = 0, \\ \dfrac{1}{x^{|n|}}, & n < 0, \\ x \cdot x^{n-1}, & n > 0. \end{cases}$$

8.14. Вивести на екран перші n рядків трикутника Паскаля. Реалізувати рекурсивний і нерекурсивний варіанти.

8.15. Для перших n чисел послідовності чисел Фібоначчі F_n переконатися у справедливості формули:

$$F_{n+1} = \sum_{k=0}^{[n/2]} C_{n-k}^{k},$$

де C_m^k – біноміальні коефіцієнти.

8.16. Визначити рекурсивні функції:

a) обчислення суми цифр;
b) кількості цифр;
c) максимальної цифри;
d) мінімальної цифри;
e) перевірки входження заданої цифри до натурального числа.

8.17. Дано послідовність натуральних чисел, що завершується числом 0 (вважати, що 0 у послідовність не входить). Визначити рекурсивні функції для знаходження:

a) максимального числа серед членів послідовності;
b) мінімального числа серед членів послідовності;
c) суми всіх членів послідовності.
d) суми додатних членів послідовності;
e) суми від'ємних членів послідовності;
f) найбільшого серед додатних членів послідовності;
g) найменшого за модулем серед від'ємних членів послідовності.

8.18. Дано перший член і різницю арифметичної прогресії. Визначити рекурсивні функції для знаходження:

a) n -го члена арифметичної прогресії;
b) суми n членів арифметичної прогресії.

8.19. Дано перший член і знаменник геометричної прогресії. Визначити рекурсивні функції для знаходження:

a) n-го члена геометричної прогресії;
b) суми n членів геометричної прогресії.

8.20. Задані натуральні числа a, c, m. Описати рекурсивну функцію за правилом

$$f(m) = \begin{cases} m, 0 \le m \le 9, \\ g(m) \cdot f(m - 1 - g(m)) + m. \end{cases}$$

де $g(m)$ – залишок від ділення $a \cdot m + 10$ на 10.

8.21. Визначити рекурсивні функції:

a) перевірки заданого рядка на симетричність;
b) побудови рядка, інвертованого відносно до заданого;
c) заміни у вихідному рядку всіх входжень даного символа даним рядком;
d) перевірки, чи є один рядок початком іншого;
e) перевірки на входження одного рядка в інший.

8.22. Скласти програму додавання "у стовпчик" двох чисел, записаних у вигляді рядків, що є позиційними записами цих чисел у десятковій системі числення. У програмі описати та використати такі підпрограми:

a) `GetDigit(c)` – повертає цифру, що відповідає символу `c`, і -1, якщо c не є символом-цифрою;
b) `GetSymbol(d)` – повертає символ, що відповідає цифрі `d`;
c) `AddDigit(n1, n2, p, n)` – підпрограма додавання двох цифр `n1`, `n2` з урахуванням перенесення `p` та отриманням останньої цифри результату в `n`;
d) `AddColumn(S, S1, S2)` – підпрограма додавання двох рядків `S1` та `S2` у стовпчик із записом результату в рядку `S`.

8.23. Скласти програму множення "у стовпчик" двох чисел, записаних у вигляді рядків, що є позиційними записами цих чисел у десятковій системі числення. У програмі описати та використати підпрограми:

a) `GetDigit(c)` – повертає цифру, що відповідає символу `c`, і -1, якщо `c` не є символом-цифрою;
b) `GetSymbol(d)` – повертає символ, що відповідає цифрі `d`;
c) `MulDigit(n1, n2, p, n)` – множення двох цифр `n1`, `n2` з урахуванням перенесення `p` та отриманням останньої цифри результату в `n`;
d) `MulStrChar(S, c)` – підпрограма множення рядка `S` на символ `c`;
e) `AddStr(S, S1, S2, n)` – підпрограма додавання двох рядків `S1` і `S2` у стовпчик зі "зсувом" другого рядка на `n` позицій ліворуч із записом результату в змінній `S`.

8.24. Даний набір символів і словник, що містить набір слів. Побудувати всі можливі слова з цього набору.

8.25. Розв'язати задачу 3.39 використовуючи функції генератори для послідовностей.

8.26. Розв'язати задачу 3.44 використовуючи функції генератори для визначення поточного члена послідовності Фібоначчі та функцію, що визначає чи є число простим.

8.27. Розв'язати задачу 3.47 використовуючи функції генератори для послідовностей.

8.28. Розв'язати задачу 3.62 використовуючи функції генератори для послідовностей.

8.29. Декоратор, що модифікує функцію, що її значення потрапляє у заданий проміжок.

Декоратор, що перевіряє аргументи функцій.

Декоратор, що модифікує дійсні аргументи функцій цілими

TODO: Додати задачі на рекурсію, генератори та декоратори.

§9 ФАЙЛИ

Файли

Файл (англ. file – папка) – це іменований блок інформації (послідовність байтів), який зберігається на носії інформації.

Файл є найменшою одиницею збереження інформації на носії. Файл має такі ознаки:

- фіксоване ім'я (назва файла) – послідовність символів, що однозначно характеризує файл;
- певне логічне зображення (що визначається типом інформації, що міститься у файлі) і відповідні йому операції читання/запису;
- розмір файла (характеризується розміром інформації, що в ньому міститься).

Роботу з файлами можна розділити на три основних етапи:

- Відкриття файлу;
- Робота з інформацією, що стосується файлу (читання/запис);
- Закриття файлу.

При роботі з файлами розрізняються **двійкові** (або бінарні) і **текстові** файли. Двійкові файли відкриваються як послідовність байтів. При цьому відповідальність за коректну робота з даними повністю покладається на програму, що використовує цей файл. Текстові файли відкриваються як послідовність рядків (символів), що міститься у файлі. При цьому фізичний рядок у файлі відповідає рядковому літералу у програмі. Створювати, змінювати та опрацьовувати двійкові файли можна як правило лише за допомогою спеціальної програми. З текстовими файлами можна працювати за допомогою будь-якого текстового редактора.

Відкриття файлу

Відкриття файлу здійснюється інструкцією

```
f = open(file_name, mode)
```

де `f` – ім'я файлової змінної, `file_name` – ім'я файла, що відкривається, `mode` – режим роботи з файлом – це рядковий літерал, що будується за допомогою значень наведених у таблиці.

Таблиця 9.1. Режими роботи з файлом

Режим	Опис
`"r"`	Відкриття для читання. Якщо файла з таким ім'ям не існує, то інструкція породжує помилку (типове значення).
`"w"`	Відкриття для запису. Якщо файл з таким ім'ям вже існує, то цей файл буде перезаписаний новим.
`"x"`	Відкриття для запису. Якщо файл з таким ім'ям вже існує, то інструкція породжує помилку.
`"a"`	Відкриття для додавання даних у кінець файлу.
`"+"`	Відкриття на читання та запис.
`"b"`	Відкриття у двійковому режимі
`"t"`	Відкриття у текстовому режимі (типове значення).

Значення режиму `mode` отримується об'єднанням значень наведених вище. Наприклад, `"rb"` – відкриття існуючого двійкового файлу для читання. `"r"` і `"t"` – це типові значення, їх можна опускати при формуванні режиму `mode`. Типове значення параметра `mode` – це `"rt"`.

Інструкція `open` ототожнює вміст файла `file_name` зі змінною `f`. Подальша робота з файлом відбувається лише через цю змінну. Крім цього інструкція `open` блокує файл для змін іншими програмами, щоб уникнути конфліктних ситуацій.

Закриття файлу

Як було сказано вище, робота з файлом відбувається через файлову змінну, що зберігає свої дані у оперативній пам'яті. Для того, щоб зберегти результат роботи з файлом на носій інформації, файл необхідно закрити. Крім цього операція закриття файлу повідомляє операційній системі, що файл розблокований і може використовуватися (для зміни) іншими програмами.

Для того, щоб закрити файл використовується інструкція

```
f.close()
```

де `f` – ім'я файлової змінної, пов'язаної з файлом.

Операції з файлами

Операція відкриття файлу породжує спеціальний об'єкт (що належить файловій змінній), що називається **маркером**. Маркер вказує на поточну позицію у файлі, тобто місце з якого відбувається читання чи запис даних.

Під час відкриття файлу у режимі `"a"` (додавання) маркер встановлюється у позицію після останнього запису. Для інших режимів маркер встановлюється

на найперший запис. Будь-яка операція читання/запису змінює поточну позицію маркера.

Нехай `f` – файлова змінна, яка асоційована з файлом відкритим у відповідному режимі. Тоді

Таблиця 9.2. Операції з файлами

Операція	Опис
`f.read()`	Повертає весь вміст файла.
`f.read(n)`	Читає `n` байтів з поточного положення маркера.
`f.readline()`	Читає та повертає поточний рядок з текстового файлу.
`f.readlines()`	Повертає список всіх рядків текстового файлу.
`f.write(s)`	Записує значення змінної `s` у файл.
`print(s, file = f)`	Записує значення змінної `s` у файл.
`f.writelines(lineslist)`	Записує список рядків `lineslist` у текстовий файл.
`f.tell()`	Повертає значення поточної позиції маркера у файлі.
`f.seek(n)`	Встановлює маркер у позицію `n` у файлі.

Текстові файли

Файлова змінна для текстового файлу – це колекція його рядків. Отже, текстовий файл можна проходити по рядках, використовуючи цикл **`for`**.

Нехай `f` – файлова змінна відкритого для читання текстового файлу. Тоді інструкція

```
for line in f:
    process_iteration
```

виконує `process_iteration` для всіх рядків файлу `f`.

Розглянемо кілька прикладів, що ілюструють роботу з текстовими файлами.

Приклад 9.1. У текстовому файл містить набір чисел, кожне з яких записано з нового рядка. Потрібно знайти суму цих чисел.

Розв'язок. Для розв'язання задачі, у програмі потрібно відкрити файл у текстовому режимі для читання і пройти по його колекції рядків.

Отже програма буде мати вигляд

```
f_name = input("Введіть ім'я файлу ")
f = open(f_name) # Відкриваємо текстовий файл для читання
suma = 0
for line in f:          # Пробігаємо файл f по рядках
    suma += float(line) # Перетворюємо line у дійсне число
print("Сума чисел записана у файлі = {}".format(suma))
f.close()               # Не забуваємо закривати файл!
```

Для перевірки роботи програми створимо текстовий файл з іменем `123.txt` який розмістимо у тій же папці, що і файл з програмою. Заповнимо цей текстовий файл деякими числами, наприклад, таким чином:

```
1
3
12
5
34
```

Запустимо виконання програми. На запит введемо ім'я вищестореного файлу. Тоді результатом виконання цієї програму буде

```
Введіть ім'я файлу 123.txt
Сума чисел записана у файлі = 55.0
```

Вище, у параграфі «Обробка виключень» було розказано про менеджер контексту. Робота з файлами, це один з випадків, коли використання контексту є більш ніж обґрунтованим. Змінимо вищенаведений код програми еквівалентним йому з використанням менеджеру контексту

```
f_name = input("Введіть ім'я файлу ")
suma = 0

with open(f_name) as f:      # Відкриваємо файл для читання за
                             # допомогою менеджеру контексту
    for line in f:           # Пробігаємо файл f по рядках
        suma += float(line) # Підраховуємо суму
# закриття файлу буде здійснено автоматично,
# щойно відбудетьс вихід з контесту

print("Сума чисел записана у файлі = {}".format(suma))
```

Приклад 9.2. Згенерувати текстовий файл, що буде містити послідовність чисел Фібоначчі, що обмежена заданим числом.

Розв'язок.

```
N = int(input("Задайте N "))
f_name = input("Введіть ім'я файлу для запису ")
f = open(f_name, "w")  # Створюємо текстовий файл для запису
F1 = F2 = 1
print(F2, file=f)      # Записуємо F2 у файл f
while F1 <= N:
    F1, F2 = F1 + F2, F1
    print(F2, file=f)  # Записуємо F2 у файл f
f.close()  # Щоб зберегти зміни у файлі
           # не забудьмо закрити файл!
```

Приклад 9.3. Дано текстовий файл, що містить послідовність слів. Крім цього задано пару слів `w1` та `w2`. Замінити у файлі усі входження слова `w1`, словом `w2`.

Розв'язок. Обробку файлу здійснимо у кілька кроки. Спочатку відкриємо файл та зчитаємо з нього дані у список рядків (після чого обов'язково закриємо файл). Далі, у отриманому списку рядків здійснимо заміну всіх входжень слова `w1` словом `w2`. І нарешті, на останньому кроці перезапишемо початковий файл оновленими даними.

```
f_name = input("Введіть ім'я файлу ")
w1 = input("Задайте w1 ")
w2 = input("Задайте w2 ")

f = open(f_name) # Відкриваємо текстовий для читання
f_lines = f.readlines()  # Зчитуємо рядки файла f у f_lines
f.close()        # Закриваємо файл, щоб перезаписати його

f = open(f_name, "w") # Перезаписуємо файл
for s in f_lines:     # Пробігаємо список усіх рядків
    s1 = s.replace(w1, w2) # Будуємо рядок s1 по рядку s у
                           # якому слова w1 замінено на w2
    print(s1, file = f, end = "")  # Записуємо s1 у файл f
f.close()
```

Для тестування програми створимо файл `1.txt` з таким вмістом

```
1 112 change 33 44
2 11 223 change 44
3 11 22 334 44
4 change 22 33 445
```

```
5 11 252 33 44
6 11 change 353 44
```

де слово `change` планується замінити іншим словом, наприклад, `replaced`. Запустимо програму та введемо дані таким чином:

```
Введіть ім'я файлу 1.txt
Задайте w1 change
Задайте w2 replaced
```

Відкривши файл `1.txt`, переконаємося, що всі входження слова `change` були замінені словом `replaced`:

```
1 112 replaced 33 44
2 11 223 replaced 44
3 11 22 334 44
4 replaced 22 33 445
5 11 252 33 44
6 11 replaced 353 44
```

Бінарні файли і сереалізатори

Обробка бінарних файлів без додаткових засобів є досить не простою задачею, оскільки запис і читання з файлу потрібно проводити у двійковому форматі (як послідовність байтів).

Розглянемо роботу з бінарними файлами за допомогою спеціалізованих бібліотек які дозволяють зберігати та завантажувати складні об'єкти Python, зокрема списки, словники, користувацькі об'єкти тощо.

Модуль pickle

Практично будь-який об'єкт можна зберегти на диску (сереалізувати) у будь-який момент його існування, а пізніше відновити його з диску (десереалізувати). Модуль `pickle` реалізує потужний алгоритм сереалізації та десереалізації об'єктів Python.

Перш ніж користуватися можливостями модуля, його необхідно підключити інструкцією

```
import pickle
```

Відкриття та закриття файлу здійснюється стандартним чином. При відкритті файлу інструкцію `open` для читання чи запису, значення `mode` має бути `'wb'` або `'rb'` відповідно.

Щоб сереалізувати об'єкт `obj` у відкритому для запису файлі `f` треба викликати інструкцію

```
pickle.dump(obj, f)
```

Для десереалізації даних з відкритого для читання файлу `f` треба скористатися інструкцією

```
obj = pickle.load(f)
```

При цьому дані будуть записані у змінну `obj`.

Ніколи не завантажуйте з допомогою модуля `pickle` дані з ненадійних джерел. Це може призвести до незворотних наслідків з даними на вашому комп'ютері.

Приклад 9.4. Нехай задано деякий об'єкт з даними (наприклад, словник). Потрібно сереалізувати цей об'єкт у файл з допомогою модуля `pickle`. Відновити дані з файлу і вивести на екран.

Розв'язок.

```
import pickle
# Дані записані у вигляді словника
data = {
    'a': [1, 2.0, 3, 4 + 6j],
    'b': ("String", "Hello"),
    'c': {None, True, False}
}

f_name = input("Введіть ім'я файлу ")
f = open(f_name, "wb")     # Відкриваємо файл для читання
pickle.dump(data, f)       # Сереалізація даних у файл
f.close()                  # Закриваємо файл

f = open(f_name, "rb")     # Відкриваємо файл для читання
data_new = pickle.load(f) # Десереалізуємо дані у data_new
f.close()
print(data_new)
```

Запустимо програму на виконання. На запит програми введемо ім'я файлу `data.pickle`.

```
Введіть ім'я файлу data.pickle
```

```
{'b': ('String', 'Hello'), 'c': {False, True, None}, 'a': [1,
2.0, 3, (4+6j)]}
```

Звернемо увагу на те, що у папці з програмою з'явився файл `data.pickle`.

Модуль shelve

Модуль `shelve` використовують для зберігання у файлах та відновлення з них даних, що мають вигляд словника. Під час роботи з модулем `shelve`, Python створює окрім основного декілька додаткових службових файлів, що використовуються для організації доступу до даних.

Перш ніж користуватися можливостями модуля `shelve`, його необхідно підключити інструкцією

```
import shelve
```

Модуль `shelve` має власну команду для відкриття файлів.

```
d = shelve.open(filename)
```

де `filename` – ім'я файлу у якому містяться дані. При цьому змінна `d` одночасно є і файловою змінною і словником, що містить дані.

У кінці роботи потрібно закрити файл командою.

```
d.close()
```

Приклад 9.5. Нехай задано деякий словник. Потрібно зберегти цей словник у файл з допомогою модуля `shelve`. Відновити дані з файлу і вивести на екран.

Розв'язок.

```
import shelve
# Дані записані у вигляді словника
data = {
    'a': [1, 2.0, 3, 4 + 6j],
    'b': ("String", "Hello"),
    'c': {None, True, False}
}

f_name = input("Введіть ім'я файлу ")
# Створюємо/перезаписуємо файл
```

```
d = shelve.open(f_name)
for k, v in data.items():
    d[k] = v
d.close()

# Відкриваємо файл та виводимо його вмісти на екран
d = shelve.open(f_name)
for k, v in d.items():
    print(k, v)
d.close()
```

Запустимо програму на виконання. На запит програми введемо ім'я файлу `data`.

```
Введіть ім'я файлу data
c {False, True, None}
b ('String', 'Hello')
a [1, 2.0, 3, (4+6j)]
```

Звернемо увагу на те, що у папці з програмою з'явився файл `data.dat` – при створенні файлу йому автоматично було додано розширення `dat`.

Робота з файловою системою

Файлова система визначає спосіб організації файлів на носіях даних. У сучасних операційних системах файлова система на логічному рівні є ієрархією каталогів (також використовують терміни тека, папка) у вигляді дерева. Кожен каталог може містити файли або підкаталоги. Кожна програма має поточний каталог. Як правило, типовим значенням поточного каталогу є каталог з якого запускається програма.

Для роботи з файловою системою використовується стандартний модуль `os`. Розглянемо основні функції модуля `os` для роботи з файловою системою.

Таблиця 9.3. Основні операції роботи з файловою системою.

Операція	Опис
`os.listdir(path='.')`	Повертає список файлів та підкаталогів у каталозі `path`.
`os.walk(top)`	Генератор-функція, що повертає послідовність кортежів для всіх підкаталогів дерева, що починається з каталогу `top`. Для кожного підкаталогу (та для самого `top`) повертається кортеж (`dirpath`, `dirnames`, `filenames`), де `dirpath` – каталог, `dirnames` – список

	підкаталогів каталогу `dirpath`, `filenames` – список файлів каталогу `dirpath`.
`os.getcwd()`	Повертає поточний каталог.
`os.chdir(path)`	Робить поточним каталог `path`.
`os.mkdir(path)`	Створює каталог `path`.
`os.rmdir(path)`	Видаляє каталог `path`. Каталог повинен бути порожнім.
`os.remove(path)`	Видаляє файл `path`.
`os.path.exists(path)`	Перевіряє, чи існує каталог (файл) `path`.
`os.path.getmtime(path)`	Повертає дату та час останньої зміни файлу (каталогу) `path`
`os.path.getctime(path)`	Повертає дату та час створення файлу (каталогу) `path`
`os.path.getsize(path)`	Повертає розмір файлу `path`
`os.path.isdir(path)`	Повертає **`True`** якщо `path` є каталогом
`os.path.isfile(path)`	Повертає **`True`** якщо `path` є файлом
`os.path.normpath(path)`	Повертає `path` у «стандартному» вигляді.
`os.path.join(path, *paths)`	Об'єднує декілька частин шляху та імені файлу `paths` у єдиний шлях `path`. При цьому символи-розділювачі у шляху вставляються/змінюються у відповідності до вимог операційної системи.
`os.path.split(path)`	Розбиває шлях на дві частини: каталог та ім'я файлу.
`os.path.splitext(path)`	Розбиває шлях на дві частини: каталог плюс початок імені файлу та розширення імені файлу.

Задачі для самостійної роботи

9.1. У текстовому файлі записана непорожня послідовність дійсних чисел, які розділяються пропусками. Визначити функції для обчислення:

a) суми компонент файла;
b) кількості від'ємних компонент файла;
c) останньої компоненти файла;
d) найбільшого із значень компонент файла;
e) найменшого із значень компонент файла з парними номерами;
f) суми найбільшого і найменшого із значень компонент;
g) різниці першої й останньої компоненти файла;
h) кількості компонент файла, які менші за середнє арифметичне всіх його компонент.

9.2. Дано текстовий файл, компоненти якого є цілими числами. Скласти

підпрограми для обчислення:

a) кількості парних чисел серед компонент;
b) кількості квадратів непарних чисел серед компонент;
c) різниці між найбільшим парним і найменшим непарним числами з компонент;
г) кількості компонент у найдовшій зростаючій послідовності компонент файла.

9.3. Описати підпрограму, яка утворює текстовий файл із 9 рядків, у першому з яких одна літера '1', у другому – дві літери '2', ... , у дев'ятому – дев'ять літер '9'.

9.4. Описати підпрограму, яка за заданою послідовністю символів формує текстовий файл із рядками по 40 літер (в останньому рядку літер може бути й менше).

9.5. У текстовому файлі F записана послідовність цілих чисел, які розділяються пропусками. Побудувати файл G, який містив би всі компоненти файла F:

а) які є невід'ємними;
б) які є парними числами;
б) які діляться на 3 і на 5;
в) які є точними квадратами;
г) записані у зворотному порядку;
ґ) за виключенням повторних входжень одного й того самого числа.

9.6. Дано текстовий файл, що містить принаймні один непорожній рядок. Описати підпрограми:

a) виведення усіх рядків файла;
b) виведення рядків, які містять більше 60 символів;
c) підрахунку кількості порожніх рядків;
d) пошуку найдовшого рядка;
e) яка обчислює максимальну довжину рядків текстового файла.

9.7. Дано текстовий файл, що містить принаймні один непорожній рядок. Описати підпрограму, що визначає, скільки рядків текстового файла:

a) починаються із заданого символа;
b) закінчуються заданим символом;
c) починаються й закінчуються одним і тим самим символом;
d) складаються з однакових символів.

9.8. Дано текстовий файл F, що містить принаймні один непорожній рядок. Описати підпрограму, яка переписує у текстовий файл G усі рядки текстового файла F:

a) із заміною в них символа '0' на '1' і навпаки;
b) в інвертованому вигляді;
c) вставляючи до початку кожного рядка його порядковий номер у файлі F.

9.9. Скласти програму пошуку усіх входжень заданої послідовності символів у текстовий файл. Результатом виконання програми має бути список,

що складається з кортежів вигляду `(номер_рядка, позиція_у_рядку)`, де `номер_рядка` – номер рядка, у якому міститься заданий фрагмент.

9.10. Дано текстовий файл. Групи символів, які відокремлені пропусками (одним або декількома) і не містять пропусків усередині, будемо називати словами. Скласти підпрограми для:

a) знаходження найдовшого слова у файлі;
b) визначення кількості слів у файлі;
c) вилучення з файла зайвих пропусків та всіх слів, що складаються з однієї букви;
d) вилучення всіх пропусків на початку рядків, у кінці рядків та між словами (окрім одного);
e) вставки пропусків у рядки рівномірно між словами так, щоб довжина всіх рядків (якщо в них більше 1 слова) була 80 символів та кількість пропусків між словами в одному рядку відрізнялась не більше, ніж на 1 (вважати, що рядки файла мають не більше, ніж 80 символів).

Результат записати у файл H.

9.11. Відомості про автомобіль складаються з його марки, номера й прізвища власника. Дано файл, який містить відомості про декілька автомобілів. Скласти процедури знаходження:

a) прізвищ власників автомобілів даної марки;
b) кількості автомобілів кожної марки.

9.12. Багаж пасажира характеризується номером пасажира, кількістю речей та їхньою загальною вагою. Дано файл пасажирів, який містить прізвища пасажирів, та файл, який містить інформацію про багаж декількох пасажирів (номер пасажира – це номер запису у файлі пасажирів). Скласти процедури для:

a) знаходження пасажира, у багажу якого середня вага однієї речі відрізняється не більше, ніж на 1 кг від загальної середньої ваги речей;
b) визначення пасажирів, які мають більше двох речей, і пасажирів, кількість речей у яких більша, ніж середнє число речей;
c) видачі відомостей про пасажира, число речей у багажі якого не менше, ніж у будь-якому іншому багажі, а вага речей небільша, ніж у будь-якому іншому багажі із цим же числом речей;
d) визначення, чи мають хоча б два пасажири багажі, які не відрізняються кількістю речей та відрізняються вагою не більше, ніж на 1 кг (якщо такі пасажири є, то показати їхні прізвища);
e) визначення пасажира, багаж якого складається з однієї речі, вагою не менше 30 кг.

9.13. Дано файл, який містить відомості про книжки. Відомості про кожну книгу - це прізвище автора, назва та рік видання. Скласти процедури пошуку:

a) назв книг даного автора, виданих із 1960 р.;
b) книг із заданою назвою. Якщо така книжка є, то надрукувати прізвища авторів та рік видання.

9.14. Дано файл, який містить номери телефонів співробітників установи:

вказуються прізвище співробітника, його ініціали та номер телефону. Визначити процедуру пошуку телефону співробітника за його прізвищем та ініціалами.

9.15. Дано файл, який містить відомості про кубики: розмір кожного кубика (довжина ребра у см), його колір (червоний, жовтий, зелений, синій) та матеріал (дерев'яний, металевий, картонний). Скласти процедури пошуку:

a) кількості кубиків кожного з перелічених кольорів та їхній сумарний об'єм;
b) кількості дерев'яних кубиків із ребром 3 см та кількості металевих кубиків із ребром, більшим за 5 см.

9.16. Дано файл, який містить відомості про товари, що експортуються. Вказано назву товару, країну, яка імпортує товар, та об'єм партії, що надійшла, у штуках. Скласти процедуру пошуку країн, до яких експортується даний товар, та загальний об'єм його експорту.

9.17. Дано файл, який містить відомості про іграшки: вказується назва іграшки (наприклад, м'яч, лялька, конструктор тощо), її вартість у гривнях та вікові межі для дітей, яким іграшка призначається (наприклад, для дітей від двох до п'яти років). Скласти процедури:

a) пошуку назв іграшок, вартість яких не перевищує 40 грн, призначених дітям 5-и років;
b) пошуку назв іграшок, які призначені дітям і 4, і 10 років;
c) пошуку назв найдорожчих іграшок (ціна яких відрізняється від ціни найдорожчої іграшки не більше, ніж на 50 грн);
d) визначення ціни найдорожчого конструктора;
e) визначення ціни всіх кубиків;
f) пошуку двох іграшок, які призначені дитині 3-х років, сумарна вартість яких не перевищує 20 грн;
g) пошуку конструктора ціною 22 грн, призначеного дітям від 5 до 10 років. Якщо такої іграшки немає, то занести відомості про її відсутність у файл.

§10 МОДУЛІ

Один з основних принципів у сучасному програмуванні, що дозволяє спростити розробку та підвищити універсальність програм це принцип модульності, згідно з яким програма розділяється на окремі іменовані блоки, що називаються модулями.

Модуль – функціонально завершений фрагмент програми, що оформлено у вигляді окремого файла з вихідним кодом, призначений для використання в інших програмах.

Модулі дозволяють розділяти складні задачі на дрібніші. Вони проектуються таким чином, щоб надати програмістам зручний для багаторазового використання функціонал (інтерфейс) у вигляді набору підпрограм, змінних, класів тощо. Програма, що має модульну структуру, зазвичай складається з сукупності модулів, серед яких виділяють головний модуль.

Отже, модулі виконують принаймні дві важливі функції:

- Повторне використання коду без необхідності явного його дублювання.
- Керування адресний простором – модуль це високорівнева організація програм, що дозволяє уникнути конфліктів у іменуванні змінних, функцій, класів, тощо.

У Python модуль це окремий файл, що містить вихідний код – змінні, підпрограми, класи, алгоритми тощо. Отже, створення власного модуля нічим не відрізняється від написання звичайної програми в окремому файлі. Всі об'єкти модуля можуть бути використані в головній програмі або інших модулях. Для цього їх потрібно імпортувати. Можна імпортувати модуль повністю або його окремі об'єкти. Імпорт проводиться з використанням імені модуля. Ім'я модуля – це ім'я файлу без розширення `".py"`.

Імпорт та використання модулів

Імпорт модуля

До цього моменту ми вже неодноразово використовували функції з різних модулів і вже знаємо, що імпорт модуля здійснюється за допомогою ключового слова **`import`**:

```
import modul
```

тут `modul` – ім'я модуля.

Як правило інструкцію імпорта модуля розміщують на початку програми.

Імпорт модуля фактично запускає виконання програми, що міститься у модулі. Тому не рекомендується імпортувати модулі отримані з ненадійних джерел. Це може пошкодити дані на вашому комп'ютері.

Під час такого імпорту, ми отримуємо доступ до всіх елементів імпортованого модуля. Кожен модуль утворює власний простір імен, що є глобальною областю видимості для всіх визначених у ньому об'єктів. Для того, щоб об'єкти цього модуля не потрапили у конфлікт з іншими глобальними іменами або іншими модулями, під час доступу до елементів модуля потрібно використовувати префікс, що складається з імені модуля разом з крапкою, наприклад

```
modul.func1()
```

тут `modul` – ім'я модуля, `func1` – функція, що міститься у цьому модулі.

```
import math
print(math.pi)
```

Однією інструкцією **`import`** можна імпортувати кілька модулів. У такому разі імена модулів перелічуються через кому:

```
import modul_1, modul_2, modul_3
```

Проте такий підхід не є рекомендованим, оскільки це знижує читабельність коду. Порівняйте візуально вищенаведену інструкцію імпорта з таким

```
import modul_1
import modul_2
import modul_3
```

Використання псевдонімів

Якщо ім'я модуля занадто довге або таке, що може призвести до конфлікту з іншими об'єктами програми то для нього можна створити псевдонім (аліас) за допомогою ключового слова **`as`**:

```
import math as m
```

При цьому всі елементи цього модуля стають доступними у програмі через ім'я `m`, а не через `math`:

```
print(m.pi) # замість префікс math вказується m
```

Фактично створення аліасу модуля, це зміна імені модуля при імпорті.

Імпорт окремих елементів модуля

У Python є можливість підключення не всього модуля, а його окремих елементів. Для цього використовується інструкція **from**, що має такий синтаксис

```
from modul import obj
```

тут `modul` – ім'я модуля, `obj` – об'єкт (змінна, функція, клас тощо) що міститься у цьому модулі.

У разі імпорту з використанням інструкції **from**, об'єкт доступний за іменем `obj` (без префіксу, на відміну від випадку імпорту всього модуля). Наприклад

```
from math import pi
print(pi)       # префікс math не вказується
```

Можна одночасно імпортувати кілька об'єктів з одного модуля. Тоді імпортовані об'єкти перелічуються через кому

```
from math import pi, e, factorial
```

Для імпорту всіх об'єктів модуля використовується інструкція

```
from modul import *
```

Відмінність від стандартного імпорту модуля полягає у тому, що всі об'єкти модуля доступні за своїми іменами без префіксу (без імені модуля).

```
from math import *
print(pi)           # префікс math не вказується
print(factorial(4)) # префікс math не вказується
```

Звернемо увагу, що під час такого способу імпорту об'єктів з модулів можливе перекриття імен, якщо два модулі надають для імпорту одне і те ж ім'я об'єкту.

Під час імпорту окремих елементів, з використанням інструкції **from**, для них можна створювати псевдоніми за допомогою ключового слова **as**. Наприклад,

```
from math import factorial as f
print(f(4))
```

Також використанні псевдонімів вирішує проблему імпорту об'єктів з різних модулів, що мають однакові імена. Імпорт об'єктів модуля з використанням інструкції **from** робить імпортовані об'єкти доступними лише для читання.

Повторний імпорт модулів

Потрібно пам'ятати, що модуль завантажується лише один раз під час виконання програми, незалежно від того, скільки разів інструкція для його імпорту трапляється у програмі. Завантаження відбувається при виконанні першої інструкції імпорту і всі наступні операції імпорту цього модуля будуть повертати вже завантажений об'єкт модуля, навіть якщо сам модуль при цьому було змінено.

Тому, якщо необхідно повторно імпортувати модуль, використовують інструкцію `reload()` модуля `importlib`:

```
from importlib import reload
reload(modul)
```

Створення модулів

Як вже стало зрозуміло, створення власного модуля не вимагає від програміста якихось додаткових зусиль. Проте існує кілька правил, яких потрібно дотримуватися при створенні власних модулів:

- Під час іменування модуля потрібно дотримуватися тих же правил, що і під час створення змінних – можна використовувати лише латинські літери, цифри та символ "_", причому першим символом у імені модуля має бути не цифра.
- Не можна називати модуль так як вбудовані функції чи ключові слова.
- Розташовувати модуль потрібно у файловій системі так, щоб до нього був доступ для імпорту з вашої програми або інших модулів, що його використовують.

Розглянемо приклад створення модуля, що описує дві функції – піднесення числа до квадрату та кубу. Створимо файл з кодом Python, з іменем `my_module.py`:

```
def my_pow2(x):
    return x ** 2

def my_pow3(x):
    return x ** 3
```

Модуль `my_module` готовий. Використаємо його у нашій програмі. Створимо інший файл з іменем `main.py`, що буде містити такий код

```
import my_module  # імпорт модуля

res2 = my_module.my_pow2(2.0) # використання функції з модуля
res3 = my_module.my_pow3(2.0) # використання функції з модуля
```

Головний модуль програми

Головним називається модуль з якого починається виконання програми. На відміну від імпортованих, ім'я яких збігається з іменем файлу у якому він міститься, головний модуль має зарезервоване ім'я "`__main__`". Це ім'я міститься у спеціальній змінній `__name__`, яка створюється під час імпорту або запуску для кожного модуля.

Як було зазначено раніше, під час імпорту модуля, його код повністю виконується. Тому, аналіз імені модуля дозволяє виконувати або не виконувати певні набори інструкцій в залежності від того чи є модуль головним. Наприклад, у випадку виконання інтерпретатором такого коду

```
if __name__ == "__main__":
    process_some_code
```

набір інструкцій `process_some_code` буде виконано лише у тому випадку, якщо програма виконується як головний модуль, а не імпортований.

Розташування модулів у файловій системі.

Python поширюється з бібліотекою стандартних модулів, що налічує понад 200 модулів. Вони забезпечують підтримку таких задач як інтерфейс до операційної системи, керування об'єктами, робота в мережі, інтернет, графічний інтерфейс тощо.

Стандартні модулі вбудовані у інтерпретатор. Для їх використання потрібно лише їх імпортувати у програмі командою `import`.

Крім стандартних, існує ціла низка додаткових спеціалізованих модулів, які призначено для розв'язання різноманітних задач. Такі модулі потрібно додатково встановлювати. Процедура встановлення такого модуля залежить

від операційної системи, що використовується і, як правило, описується у документації модуля.

Під час імпорту модуля, інтерпретатор шукає файл з модулем у папках у такій послідовності

- Серед стандартних вбудованих модулів (каталоги Python).
- У папці програми, що імпортує модуль.
- У папках, що зазначені у змінній оточення PYTHONPATH операційної системи.

Список цих папок міститься у змінній `sys.path`.

```
>>> import sys
>>> print(sys.path)
['D:\\Repository\\Test', 'C:\\IDE\\python\\python35.zip',
'C:\\IDE\\python\\DLLs', 'C:\\IDE\\python\\lib',
'C:\\IDE\\python', 'C:\\IDE\\python\\lib\\site-packages']
```

Отже, є можливість під час виконання вашої програми розширити список каталогів, де інтерпретатор буде шукати модулі для імпорту, додавши у змінну `sys.path` новий каталог

```
>>> sys.path.append('C:/my_modules')
```

Також можна задати або доповнити змінну оточення PYTHONPATH у операційній системі шляхами до каталогів де знаходяться користувацькі модулі.

Пакети

Пакет – це спосіб структурування простору імен модулів на базі файлової системи.

Отже, якщо модулі це файли, що містять вихідний код, то пакети – це каталоги, що містять модулі. Пакети формуються у вигляді ієрархії папок, що містять модуль.

Пакети дозволяють структурувати колекції модулів, що утворюють великі бібліотеки – пакетний імпорт робить код більш читабельним і значно пришвидшує пошук.

Застосування пакетів робить безпечним використання імен модулів у багатомодульних пакетах.

Структура пакета

Розглянемо приклад пакета **TCP** – кореневий каталог **TCP** (Рис.10.1). У ньому знаходиться два підкаталоги **Server** і **Client**. Кожен з каталогів містить файл (модуль) `__init__.py` Цей файл може бути або порожнім або містити код ініціалізації пакета – код котрий виконується під час імпорту пакету. У ранніх версіях Python файл `__init__.py` вказував, що каталог, який його містить, є пакетом. Починаючи з версії 3.3 файл `__init__.py` є необов'язковим.

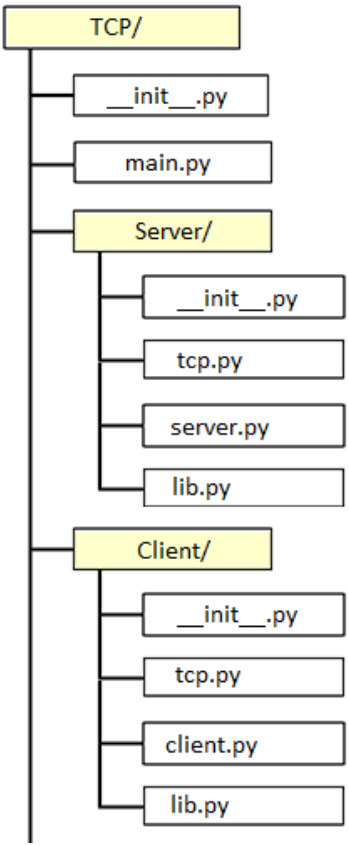


Рис. 10.1. Структура пакета TCP.

Для того, щоб імпортувати модуль, потрібно вказати повний шлях (з урахуванням вкладення всіх каталогів пакета) до модуля, що імпортується

через символ "." (крапку). Наприклад, імпорт модуля `lib.py` з пакету **Client** буде виглядати таким чином:

```
>>> import TCP.Client.lib
```

При цьому посилання на функцію у програмі також має бути повним:

```
>>> TCP.Client.lib.connect()
```

Такий спосіб часто буває не зручним з огляду на його громіздкість, тому рекомендується при імпорті використовувати вищеописаний механізм псевдонімів (аліасів). Наприклад,

```
>>> import TCP.Client.lib as cl_lib
```

Тоді відповідно виклик функції `connect()` з імпортованого модуля буде виглядати як

```
>>> cl_lib.connect()
```

Також звернемо увагу, що у пакеті містяться два модулі з однаковими іменами – `lib.py`. Проте, зважаючи на спосіб імпорту модулів, жодного конфлікту у програмі не виникне – ці два модулі знаходяться у різних папках пакету.

Задачі для самостійної роботи

10.1. Опишіть модуль, у якому описати функції для обчислення значень елементарних математичних функцій (див. задачу 3.59). Використовуючи цей модуль, знайдіть значення арифметичного виразу:

$$\frac{\ln 4 + e^3 - \sin 3^{3/2}}{\cos 2 + \mathrm{ch} 4}.$$

10.2. Опишіть модуль роботи з квадратними матрицями та векторами. У модулі опишіть такі:
операції введення/виведення:

a) зчитування вектора з клавіатури;
b) зчитування матриці з клавіатури;
c) зчитування вектора з текстового файла;
d) зчитування матриці з текстового файла;
e) виведення вектора на екран;
f) виведення матриці на екран;
g) запис вектора у текстовий файл;

h) запис матриці у текстовий файл;

операції роботи з векторами і матрицями:

a) множення матриць;
b) множення матриці на вектор;
c) множення вектора на число;
d) множення вектора на матрицю;
e) множення i-го рядка матриці на число a;
f) перестановка i-го і j-го рядків матриці;
g) перестановка i-го і j-го стовпчиків матриці;
h) додавання до i-го рядка матриці j-й рядок, помножений на число a;
i) отримання рядка матриці;
j) віднімання вектора від всіх рядків матриці.

З використанням модуля розв'язати задачі:

a) Перетворити матрицю у верхню трикутну лінійними перетвореннями;
b) Визначити ранг матриці;
c) Обчислити визначник матриці;
d) Обчислити обернену матрицю.

10.3. Використовуючи модуль задачі 10.2, знайдіть експоненту матриці:

$$e^A = E + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots,$$

де A – задана матриця, E – одинична матриця, врахувавши при обчисленні n доданків суми у правій частині.

10.4. Описати модуль роботи з точками та відрізками на площині. Типи точки та відрізку представити у вигляді кортежів:

```
Точка    – (x, y)
Відрізок – (a, b)
```

де `a, b` – точки. Реалізувати дії над точками:

a) взяти точку t;
b) покласти точку t рівною (x, y);
c) показати точку t.

Реалізувати дії над відрізками:

a) взяти відрізок s;
b) показати відрізок s;
c) покласти відрізок s рівним a, b;
d) довжина відрізку s;
e) чи лежить точка t на одній прямій з відрізком s;
f) чи лежить точка t всередині відрізку s;
g) площа трикутника, утвореного точкою t та відрізком s.

У файлі записано послідовність точок. З використанням модуля роботи з точками та відрізками на площині знайти:

a) трикутник з найбільшою площею, утворений точками послідовності;
b) коло найменшого радіуса, всередині якого лежать всі точки послідовності;

c) відрізок, на якому лежить найбільша кількість точок послідовності;
d) коло, на якому лежить найбільша кількість точок послідовності.

10.5. Реалізувати модуль для роботи з поліномами $P_n(x)$ степеня n. Поліном реалізувати як список коефіцієнтів. У модулі реалізувати такі операції:

a) зчитати поліном $P_n(x)$;
b) показати поліном$P_n(x)$;
c) визначити значення полінома $P_n(x)$ для заданого x;
d) знайти похідну $P_n'(x)$ від полінома $P_n(x)$;
e) знайти первісну $F_n(x)$ полінома $P_n(x)$, що задовольняє умову $F_n(0) = c$, де c – задане дійсне число;
f) знайти суму двох поліномів;
g) знайти добуток двох поліномів;
h) реалізувати операцію ділення з остачею двох поліномів.

Використовуючи описаний модуль, розв'язати задачі:

a) відстань, яку пройшла матеріальна точка за час t визначається за законом $s(t) = P_n(t)$; визначити її швидкість у момент часу t;
b) швидкість руху точки в момент часу t визначається за законом $v(t) = P_n(t)$; визначити яку відстань пройде точка за відрізок часу $[t_1, t_2]$;
c) знайти похідну порядку k від полінома $P_n(x)$;
d) визначити найбільший спільний дільник двох поліномів $P_n(x)$ і $R_n(x)$;
e) визначити найменше спільне кратне двох поліномів;
f) розв'язати задачу Коші для звичайного диференціального рівняння

$$y^{(k)} = P_n(x), \qquad x \geq x_0, \;\; y(x_0) = y_0, \ldots, y^{(k-1)}(x_0) = y_0^{k-1}.$$

Список літератури та використані джерела

1. The Python Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python.org/3/tutorial/index.html.
2. Навчальні матеріали: Python [Електронний ресурс] – Режим доступу до ресурсу: http://www.matfiz.univ.kiev.ua/pages/13.
3. Орлов С. А. Технологии разработки программного обеспечения. Разработка сложных программных систем [Текст] : учеб. пособие для вузов по направлению " Информатика и вычисл. техника" / Сергей Александрович Орлов. – СПб.: Питер, 2002. – 463 с.
4. Прохоренок Н. А. Python 3 и PyQt. Разработка приложений. / Николай Анатольевич Прохоренок. – СПб: БХВ-Петербург, 2012. – 704 с.
5. Васильев А. Н. Python на примерах. Практический курс по программированию / А. Н. Васильев. – СПб.: Наука и техника, 2016. – 432 с. – (Просто о сложном).
6. Python 3 для начинающих [Електронний ресурс] – Режим доступу до ресурсу: pythonworld.ru.
7. Креневич, А.П. С у задачах і прикладах : навчальний посібник із дисципліни "Інформатика та програмування" / А.П. Креневич, О.В. Обвінцев. – К. : Видавничо-поліграфічний центр "Київський університет", 2011. – 208 с.
8. Збірник задач з дисципліни "Інформатика і програмування" / Вакал Є.С., Личман В.В., Обвінцев О.В., Бублик В.В., Довгий Б.П., Попов В.В. -2-ге видання, виправлене та доповнене –К.: ВПЦ "Київський університет", 2006.– 94 с.
9. E-Olymp [Електронний ресурс] – Режим доступу до ресурсу: www.e-olymp.com.
10. Школа программиста [Електронний ресурс] – Режим доступу до ресурсу: http://acmp.ru/
11. Абрамов С.А., Гнездилова Г.Г., Капустина Е.Н., Селюн М.И. Задачи по программированию. –М.: Наука, 1988. – 224 с.
12. Златопольский Д.М. Сборник задач по программированию. – 2-е издание, переработанное и дополненное. – СПб.: БХВ-Петербург, 2007. –240 с.: ил.
13. Пильщиков В.Н. Сборник упражнений по языку Паскаль: Учебное пособие для вузов . –М.: Наука, 1989. –160 с.
14. Проскуряков И.В. Сборник задач по линейной алгебре. 11-е издание, стереотипное. – СПб.: Лань, 2008. –480 с.
15. Вирт Н. Систематическое программирование. Введение.–М.: Мир, 1977. –184 с.
16. Вирт Н. Алгоритмы + структуры данных = программы.–М.: Мир, 1985. –406 с.

17. Калиткин Н.Н. Численные методы.– Наука, 1978. — 512 с.