

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Вступ до програмування мовою Python

Навчальний посібник

Карнаух Тетяна Олександрівна

УДК 004.4

Рецензенти: доктор технічних наук, проф. Бичков О.С.,
доктор фізико-математичних наук, проф. Пашко А.О.,
кандидат фізико-математичних наук, доц. Омельчук Л.Л.,
доктор філософії в галузі інформаційних технологій Крамов А.А.

Рекомендовано до друку вченою радою факультету комп'ютерних наук та кібернетики (протокол № 3 від 14 жовтня 2025 року).

Ухвалено науково-методичною радою Київського національного університету імені Тараса Шевченка (протокол № 01-26 від 12 січня 2026 року).

Карнаух Т. О. Вступ до програмування мовою Python: Навчальний посібник. Київ — 2025.

Для студентів факультету комп'ютерних наук та кібернетики, що вивчають програмування мовою Python.

ЗМІСТ

Вступ	7
1 Загальні поняття програмування	9
1.1 Парадигми програмування	9
1.2 Програми, дані, моделі, мови	9
1.3 Алгоритми та їхні властивості	11
1.4 Мови програмування	12
1.5 Від тексту програми до її виконання	13
1.6 IDE для мови Python	15
1.7 З історії створення мови Python	15
1.8 Офіційна документація на мову Python	16
1.9 Перша програма — «Hello, World!»	16
Контрольні запитання	17
2 Зображення інформації у комп'ютері	18
2.1 Позиційні системи числення	18
2.2 Зображення цілих чисел	21
2.3 Зображення дійсних чисел	22
2.4 Зображення символів	25
Контрольні запитання	27
Задачі та вправи	28
3 Типи даних, арифметичні дії, вирази	30
3.1 Поняття типу даних в інформатиці	30
3.2 Вбудовані числові типи даних мови Python	30
3.3 Прості арифметичні дії над числовими даними	32
3.4 Обчислення в інтерпретаторі	34
3.5 Вирази	35
3.6 Виконання арифметичних дій над числовими значеннями різних типів	38
3.7 Явне перетворення типів	39
Контрольні запитання	39
Задачі та вправи	40
4 Лексичні елементи програми мовою Python	42
4.1 Алфавіт та словник мови Python	42
4.2 Ідентифікатори та ключові слова	42
4.3 Оператори та роздільники	43
4.4 Літерали	44
4.5 Коментарі	47
4.6 Логічні рядки та спеціальні лексеми	48
Контрольні запитання	50
Задачі та вправи	51
5 Дані в пам'яті програми	53
5.1 Об'єкти, або Зображення даних у пам'яті	53
5.2 Поняття змінної в інформатиці	55
5.3 Змінні в мові Python	56
5.4 Присвоювання	61
5.5 Синтаксична структура програми мовою Python	63
5.6 Вирази, що задають форматування	63
Контрольні запитання	68
Задачі та вправи	68

6	Функції	70
6.1	Підпрограми.....	70
6.2	Погляд з висоти пташиного польоту на те, як працюють функції.....	71
6.3	Елементи організації консольного введення та виведення.....	72
6.4	Бібліотечні математичні функції та константи	76
6.5	Написання та використання власних функцій	77
6.6	Локальні змінні функцій.....	81
6.7	Тільки позиційні та тільки ключові параметри	83
6.8	Значення за умовчанням для параметрів функції	83
6.9	Перевантаження функцій	84
6.10	Способи передачі аргументів у функцію – як це буває в <i>інших мовах програмування</i>	85
6.11	Документування коду	86
6.12	Довгі функції	89
6.13	Структура консольного застосування та функціональне проектування.....	90
	Контрольні запитання	92
	Задачі та вправи	93
7	Керування порядком обчислень	96
7.1	Умови	96
7.2	Інструкція розгалуження	103
7.3	Операція розгалуження.....	105
7.4	Кілька слів про стиль коду та значення, що повертаються з функцій	106
7.5	Використання інструкції match як інструкції вибору варіантів	108
7.6	Кілька слів про тестування.....	112
	Контрольні запитання	113
	Задачі та вправи	114
8	Обробка помилок. Винятки.....	118
8.1	Обробка помилкових та неочікуваних ситуацій	118
8.2	Приклад програми з обробленням винятків	119
8.3	Ієрархія вбудованих бібліотечних класів винятків та порядок пасток	121
8.4	Більш точна діагностика помилок	122
8.5	Загальний вигляд інструкції <code>try</code> та інструкція <code>raise</code>	124
8.6	Розмотування стеку.....	127
8.7	Стратегії обробки помилкових та неочікуваних ситуацій	128
	Контрольні запитання	129
	Задачі та вправи	129
9	Організація виконуваного коду в інтерпретаторі	134
9.1	Область видимості, або область дії означення/оголошення, імені та простір імен.....	134
9.2	Логічна організація виконуваного коду в інтерпретаторі.....	135
9.3	Робота інтерпретатора зі змінними	136
9.4	Розв'язання імен, або Пошук імен.....	137
9.5	Зміна значень вільних змінних у викликах функцій	140
9.6	Час існування змінних	141
9.7	Обробка інтерпретатором функцій: резюме.....	141
	Контрольні запитання	142
	Задачі та вправи	144
10	Модулі та пакети	145
10.1	Модулі	145
10.2	Пакети.....	151

Контрольні запитання	156
11 Рекурсія	157
11.1 Рекурсивні означення	157
11.2 Рекурсивні функції мовою Python та виконання їх викликів	158
Контрольні запитання	162
Задачі та вправи	162
12 Ітерація	164
12.1 Інструкція циклу з передумовою	164
12.2 Інваріант циклу	167
12.3 Оператор присвоювання	167
12.4 Загальний вигляд інструкції циклу while у мові Python	168
12.5 Інструкція циклу for	169
12.6 Рекурентні співвідношення в циклічних алгоритмах	171
12.7 Переривання та продовження циклу – зло	176
12.8 Структурованість коду	176
Контрольні запитання	178
Задачі та вправи	179
13 Задача наближеного обчислення суми степеневого ряду	185
13.1 Програмування обчислення з використанням рекурентних співвідношень	185
13.2 Стратегії обробки помилок у математичних розрахунках	187
14 Елементи технології розроблення програм	188
14.1 Від задачі до готової програми	188
14.2 Якість програмного забезпечення	191
14.3 Фактори, що впливають на ефективність роботи з кодом	192
14.4 Деякі ознаки поганого коду	193
Контрольні запитання	195
Задачі та вправи	195
15 Послідовнісні контейнери	196
15.1 Поняття структури даних	196
15.2 Послідовнісні контейнери	196
15.3 Послідовнісні контейнери мови Python	197
15.4 Найпростіші способи утворення послідовнісних контейнерів	198
15.5 Доступ до елементів	199
15.6 Застосовні до послідовностей оператори	201
15.7 Копіювання послідовностей	202
15.8 Відрізки послідовностей	205
15.9 Операції, що застосовні до послідовностей змінюваних типів	208
15.10 Особливості ітерування послідовностями	211
15.11 Послідовності як аргументи функцій	212
15.12 Оброблення послідовнісних контейнерів: підсумки	213
15.13 Оброблення рядків	214
15.14 Системи із скінченною кількістю станів як засіб моделювання поведінки	221
Контрольні запитання	225
Задачі та вправи	226
16 Рядок виклику та обробка аргументів виклику	231
16.1 Рядок виклику	231
16.2 Обробка аргументів рядка виклику у випадку програми, написаної мовою Python	231
16.3 Стандартні потоки	232
Контрольні запитання	233

Задачі та вправи	233
17 Вбудовані засоби роботи з послідовностями	234
17.1 Ітератори	234
17.2 Наповнювання, або абстракція, списків	239
17.3 Генератори	242
17.4 Лямбда-вирази та анонімні функції.....	246
17.5 Вбудовані засоби роботи з послідовностями	248
Контрольні запитання.....	253
Задачі та вправи.....	253
18 Асоціативні контейнери	255
18.1 Поняття хеш-функції	255
18.2 Поняття ключа.....	257
18.3 Поняття асоціативного контейнера	257
18.4 Асоціативні контейнери мови Python	258
18.5 Типові операції над множинами мови Python	258
18.6 Словники в мові Python	264
18.7 Використання словників в інтерпретаторі Python	274
Контрольні запитання.....	274
Задачі та вправи.....	275
19 Функції, що здатні обробляти змінну кількість аргументів.....	280
19.1 Функції зі змінною кількістю позиційних аргументів виклику	280
19.2 Функції зі змінною кількістю ключових аргументів виклику	281
19.3 Використання функцій зі змінною кількістю аргументів	283
Контрольні запитання.....	284
Задачі та вправи.....	284
20 Поняття складності обчислень.....	285
20.1 Швидкодія (performance) та її аналіз.....	285
20.2 Емпіричний підхід.....	285
20.3 Математичний підхід.....	286
20.4 Час найгіршого випадку та час у середньому	287
20.5 Асимптотична складність.....	288
20.6 Просторова складність.....	289
20.7 Складність задачі.....	290
Контрольні запитання.....	290
Задачі та вправи.....	290
Предметний покажчик.....	292
Список використаних джерел	295

Вступ

Даний посібник створено на основі частини обов'язкового курсу програмування, який читається студентам-бакалаврам 1-го курсу факультету комп'ютерних наук та кібернетики освітніх програм «Прикладна математика» та «Системний аналіз».

Мов програмування та їх діалектів існує дуже багато, мови програмування постійно оновлюються (офіційна нова версія C++ виходить раз на три роки, для мови Python цей період вже складає не півтора роки, а тільки рік) та виникають нові. Сенсу в курсі програмування зазубрювати синтаксис тієї чи іншої мови програмування в усіх можливих дрібницях з усіма її стандартними бібліотеками вже нема (ще не встигнемо до кінця вивчити одну мову/діалект, а треба вже переходити на іншу, що може відрізнятись деякими змінами в синтаксисі чи в бібліотеках, що містять стандартний набір допоміжних підпрограм). Треба розумітися на базових концепціях (типи даних, оператори, змінні, інструкції, організація пам'яті програми тощо), що притаманні багатьом мовам програмування, уміти користуватися ними на прикладі якоїсь конкретної мови та вміти працювати з документацією на мову програмування. Саме це дасть можливість легко перейти на програмування іншою, більш новою мовою чи діалектом. Це перше, що буде розглядатися в цьому посібнику на прикладі мови Python.

Різноманітних алгоритмів на сьогодні існує дуже багато: алгоритми сортування, чисельні алгоритми, алгоритми на графах тощо. Як правило, вони зорієнтовані на розв'язання задач з певних галузей, інколи доволі далеких одна від одної; тобто достатньо спеціалізовані. З цієї точки зору, цей курс програмування, який по своїй суті є вступним/початковим, майже не буде містити алгоритмів. Але різноманітні алгоритми мають спільне: всі вони використовують певні загальні структури даних. Ось базові структури даних (способи організації даних) дещо розглядатимуться (але не більше, ніж це матиме сенс для мови Python).

На початку входження обчислювальної техніки в людське життя комп'ютери, порівняно з сучасними, були майже іграшковими за своїми можливостями (швидкодія та пам'ять), програми не мали графічного інтерфейсу, були порівняно невеликими за об'ємом та призначалися переважно для розв'язання розрахункових задач. Оскільки обчислювальні ресурси комп'ютерів тих часів були невеликі, то гостро ставали питання часу роботи програми та використовуваної пам'яті. З огляду на останнє вимогою того часу було використання швидких та ефективних алгоритмів. Швидкі алгоритми доволі часто потребують специфічної організації даних в пам'яті, тому водночас з алгоритмами розвивалися різноманітні структури даних.

Сьогодні, коли обчислювальна техніка стала дуже потужною, а програми дуже великими, для багатьох сучасних програм вузьким місцем стали не обмеження на час роботи програми чи використовувану пам'ять, а можливість людини керувати кодом: розробляти, налагоджувати та модифікувати сучасні програмні комплекси, — тобто **технологія розроблення програм**, однією з складових якої є система організації коду.

Цей посібник не пройде осторонь від елементів розроблення програмного забезпечення, хоча за браком часу повноцінно розглянути й не зможе.

Отже, у посібнику розглядатиметься:

- 1) базові елементи сучасних мов програмування високого рівня на прикладі мови Python;
- 2) вбудовані структури даних, наявні у мові Python;
- 3) елементи технології розроблення програмного забезпечення;

Також у тексті використовуватимуться певні позначки.

Знаком ◀ будемо позначати кінець прикладів.

У записах вигляду **позначення_типу(аргумент)** будемо розуміти, що замість виділених жирним курсивом частин у конкретному коді мають бути підставлені конкретні вирази, тобто що такі записи по суті задають шаблони використання синтаксичних конструкцій.

Для тексту програм (та їх фрагментів) будемо використовувати таке стилістичне оформлення.

```
print(2025)
```

Вихід відповідного коду будемо оформлювати так.

```
2025
```

У деяких випадках буде розглядатися код, що покроково виконується в інтерпретаторі (набрали порцію коду, натиснули ENTER, у відповідь інтерпретатор його виконує і наприкінці виводить результат його роботи та, якщо в кінці обчислювався вираз, то також виводиться отримане значення). Код, що виконується позначено сірим фоном, безпосередньо за ним наведено вихід, породжений в результаті його виконання.

```
'Big brown fox.'.isupper()
```

```
False
```

1 Загальні поняття програмування

1.1 Парадигми програмування

Чому саме і чи завжди можна легко перейти від програмування однією мовою до програмування іншою? Легкий перехід від однієї мови до іншої можливий завдяки спільним принципам і концепціям їх будови, якщо це так.

Парадигма програмування — сукупність ідей і понять, що визначають стиль написання комп'ютерних програм.

Одними з двох найбільш глобальних та протилежних одна одній парадигм програмування є **декларативне** та **імперативне** програмування.

Декларативний підхід визначається тим, що він стверджує, тобто декларує, певні факти. У декларативному програмуванні для отримання чогось описують його властивості (наприклад, системою рівнянь).

Імперативний підхід, навпаки, надає опис того, як це отримати, тобто, які дії треба зробити для його одержання.

Розглянемо обидва підходи на прикладі тістечка:

— тістечко має бути пухким і солодким (декларативний підхід);

— для приготування тістечка візьміть яйця, цукор, борошно, ... (імперативний підхід).

Такі мови як Basic, Pascal, Delphi, C, C++, Python відносяться до так званої імперативної парадигми. У цьому курсі розглядатиметься імперативне програмування. Декларативне та імперативне програмування — не єдині існуючі парадигми. За іншими ознаками виділяють й інші парадигми, які перетинаються або навіть вкладаються в зазначені. Наприклад, **об'єктно-орієнтоване програмування**.

1.2 Програми, дані, моделі, мови

Сучасна людина використовує комп'ютер для розв'язання різноманітних задач — від виконання простих арифметичних дій до моделювання атмосферних явищ та керування польотами в космос. Насправді все, що «вміє» комп'ютер, — це виконувати програми. Щоб комп'ютер розв'язав потрібну задачу, необхідно спочатку створити відповідну програму, а потім запустити її на виконання.

«Програма» (у перекладі з грецької «припис») — попередній опис подій або дій, що мають відбутися.

Якщо дуже-дуже спрощено, то **комп'ютерна програма** — це опис дій, які має виконати комп'ютер, щоб розв'язати деяку задачу. Комп'ютерні програми розвивалися одночасно з розвитком комп'ютерної техніки та інформаційних технологій. Існує багато означень та тлумачень цього терміну, які доповнюють та уточнюють одне одного. Серед яких є й таке.

Комп'ютерна програма — зображена в об'єктивній формі сукупність даних і команд, призначених для функціонування ЕОМ та інших комп'ютерних пристроїв з метою отримання певного результату, включаючи підготовчі матеріали, отримані під час розробки програми для ЕОМ, та породжувані нею аудіовізуальні відображення.

Програмування — це діяльність, яка полягає у створенні програм. Мета програмування — забезпечити, щоб замість людини ту чи іншу роботу виконував комп'ютер.

Усі дії комп'ютера пов'язані з обробкою даних — числових, символічних, текстових тощо. Слідуючи В.М. Глушкову, **дані** (англ. *data*) зображують (позначають) деякий **зміст**, або **інформацію**. Поняття змісту та інформації не мають чіткого означення або тлумачення. Будемо розуміти їх як знання, відомості про щось.

Приклади. Слово НЕБО записано чотирма буквами. Зміст, який позначено ними, не є чітким, але для кожної людини це щось велике над головою, блакитне вдень та чорне вночі. Уточнювати далі не будемо.

Запис 1001 позначає число — уявне поняття, яке виражає кількість. Ми сприймаємо ці дані як «тисяча й один» (предмет, рік тощо) або «тисяча й одна» (ніч, дрібниця тощо).

Коли батьки реєструють народжену дитину, вони отримують свідоцтво про народження. У ньому вказано ім'я та прізвище нової людини, дату й місце народження, імена батьків. Ці дані про людину позначають факт і деякі з обставин її появи на світ. ◀

Отже, зображення об'єктів реального світу за допомогою даних є основою будь-якої взаємодії, у тому числі й комп'ютера із «зовнішнім» світом.

Комп'ютер має розв'язувати задачі для людини. У цих задачах фігурують різноманітні об'єкти реального світу — картинки, фізичні явища, особи, технологічні процеси тощо. Щоб запрограмувати обробку даних, пов'язаних із цими об'єктами, треба спочатку **зобразити об'єкти у вигляді даних**.

Зображення об'єкта, явища або процесу за допомогою даних про нього є його **моделлю**. Узагалі, **модель** — це спрощене зображення об'єкта або явища, яке відображає його необхідні суттєві властивості. В обробці даних модель являє собою сукупність властивостей та співвідношень між ними, що відображають суттєві риси досліджуваного об'єкта, явища або процесу. Модель, як правило, містить опис не лише даних, але й їх обробки, яка відтворює реальні процеси, пов'язані з об'єктом.

Існує безліч різновидів моделей. Один і той самий об'єкт реального світу може мати декілька різних моделей, що виражають «свої» властивості, потрібні з різних точок зору на нього.

Приклади

1. Свідоцтво про народження або паспорт є дуже простою моделлю людини. Складнішою є, наприклад, медична картка, яка включає дані про фізичні та фізіологічні властивості людини, зміну їх у часі тощо. Зовсім інші дані про людину присутні в його таблиці навчання або атестаті.

2. Щоб розв'язати рівняння вигляду $ax^2 + bx + c = 0$, потрібні лише його коефіцієнти — числа a , b , c , тобто трійка цих чисел (дані) зображує рівняння.

3. Кожне сучасне підприємство має програмні системи, які автоматизують облік кадрів, заробітної платні, фінансової та виробничої діяльності. Кожна з цих систем має «свої» дані, пов'язані з підприємством, які використовуються та обробляються тільки в ній. Ці дані складають частину відповідної (кадрової, фінансової тощо) моделі підприємства. Інша частина моделі описує, як присутні в моделі дані використовуються та обробляються. ◀

Власне, дані — це деякі позначення, тобто записи, що позначають властивості об'єкта. Однак для правильного розуміння запису (відновлення позначеного ним змісту), необхідно знати, що саме позначають окремі елементи цього запису.

Система позначень деякого змісту називається **мовою**. Мова включає елементарні позначення, правила утворення складніших позначень із простіших та правила, за якими зміст і позначення відповідають одне одному. Правила утворення позначень визначають **синтаксис** мови, а правила, які задають зміст позначень, — **семантику**.

Для спілкування й мислення людина використовує так звані **природні мови** (наприклад, українську, російську, англійську тощо). Природним мовам притаманно те, що речення в них можуть мати неоднозначний зміст, що може залежати від загального контексту, інтонації тощо.

Проте в науці й техніці природні мови неефективні або недостатні, тому розроблено також спеціальні **штучні мови**. Їх застосовують насамперед для обміну інформацією між користувачем і/або прикладними процесами. Одним із класів штучних мов є **мови програмування**, призначені для запису програм. Штучні технічні мови більш стандартизовані та більш лаконічні, ніж природні; крім того ним не притаманна проблема можливості різного тлумачення змісту позначень.

Програма, записана мовою програмування, завжди має єдиний зміст, але, можливо, не той, що мався на увазі програмістом ☹.

1.3 Алгоритми та їхні властивості

Програму, призначену для виконання комп'ютером, можна розглядати як різновид такого більш загального поняття, як алгоритм.

Алгоритм — це опис порядку дій, які треба виконати, щоб розв'язати деяку задачу. Позначення дій в алгоритмі називаються **командами** або **інструкціями** (англ. *statement*).

Як правило, в алгоритмі вказано деякі **вхідні** (англ. *input data*), **результатні** (**вихідні**, англ. *output data*) та **проміжні дані** (англ. *intermediate data, temporary information*), що не є ні вхідними, ні вихідними.

Раніше до поняття алгоритму прив'язували поняття процесу, як послідовності дій, яка виконується за алгоритмом. Але на сьогодні, коли розповсюджене багатопотокове програмування, такий підхід є морально застарілим. Згідно ISO 9000:2000 під **процесом** розуміють сукупність дій, що взаємозв'язані і взаємодіють та перетворюють вхідні дані у вихідні. Існують й інші означення процесу, які уточнюють попереднє та дають погляд на процес з точки зору операційних систем.

Приклад. Розглянемо задачу «обчислити корені рівняння $ax^2 + bx + c = 0$, заданого коефіцієнтами a, b, c (за умови $a \neq 0$)». Алгоритм розв'язання цієї задачі, тобто опис визначення коренів, може мати такий вигляд (насправді не найкращий з точки зору технології)

1. Прочитати коефіцієнти a, b, c .
2. Обчислити дискримінант $d = b*b - 4*a*c$.
3. Якщо $d > 0$, то

обчислити $x1 = \frac{-b-\sqrt{d}}{2a}$, $x2 = \frac{-b+\sqrt{d}}{2a}$ та написати ці числа;

інакше, якщо $d = 0$, то

обчислити $x = \frac{-b}{2a}$ й написати це число;

інакше написати «дійсних коренів немає».

У цьому алгоритмі вхідними даними є коефіцієнти a, b, c , проміжними — дискримінант d , вихідними — два корені (можливо, один) або текст «дійсних коренів немає».

За цим алгоритмом, залежно від конкретних вхідних даних, можливе виконання однієї з трьох таких послідовностей дій.

1. Прочитати коефіцієнти, обчислити d , перевірити, що $d > 0$ (і це так), обчислити $x1, x2$ й написати ці числа.

2. Прочитати коефіцієнти, обчислити d , перевірити, що $d > 0$ (і це не так), перевірити, що $d = 0$ (і це так), обчислити x і написати це число.

3. Прочитати коефіцієнти, обчислити d , перевірити, що $d > 0$ (і це не так), перевірити, що $d = 0$ (і це не так), і написати, що дійсних коренів немає. ◀

Алгоритми мають кілька загальних властивостей: зрозумілість, результативність, однозначність, дискретність, масовість та виконуваність. Розглянемо їх.

Зрозумілість. Для виконання алгоритму завжди потрібен **виконавець**. Це може бути людина або деяка технічна система, зокрема, й комп'ютер. Наприклад, виконувати арифметичні дії, розв'язуючи квадратне рівняння (і не тільки), може людина. Але вона може перекласти цю роботу на комп'ютер, якщо створить відповідну програму й примусить комп'ютер її виконати. Так само, збирати прилади може спеціальна автоматична лінія, якщо виконує відповідну програму.

Зрозумілість алгоритму полягає в тому, що виконавець може правильно зрозуміти та виконати команди, записані в алгоритмі. Команди ж завжди записуються за допомогою певної системи позначень, тобто мови. Отже, виконавець повинен розуміти мову запису алгоритму.

Однозначність (детермінованість). Алгоритм не повинен містити команд, зміст яких можна сприйняти неоднозначно. Наприклад, якби в алгоритмі розв’язання квадратного рівняння була команда «обчислити x або написати, що коренів немає», виконавець не знав би, що ж саме йому робити. Окрім того, після виконання кожної команди виконавець має точно знати, що йому робити далі.

Дискретність. Дискретність алгоритму полягає в тому, що послідовні дії чітко відокремлені в часі. Окрім того, виконання кожної команди повинно займати обмежений проміжок часу.

Масовість. Конкретні об’єкти, до яких застосовуються дії під час виконання алгоритму, визначають конкретні задачі, які часто називаються **екземплярами задачі**. Наприклад, конкретна трійка чисел 3, 10, 2 відповідає квадратному рівнянню, яке треба розв’язати. Масовість алгоритму полягає в тому, що він застосовний до різних наборів вхідних даних, тобто до різних екземплярів задач. Найчастіше алгоритм описує не один, а деяку **множину процесів**, які відбуваються при розв’язанні всіх можливих екземплярів задач; хоча існують і алгоритми, що задають тільки один процес.

Результативність. Виконання будь-якого алгоритму має приносити його виконавцю або іншій особі відчутні результати. Наприклад, «корені рівняння визначено», «прилад зібрано» тощо.

Виконуваність та скінченність. Алгоритм має бути таким, щоб на кожному екземплярі задачі його можна було **виконати до кінця** (й отримати результат). Кожен процес, заданий алгоритмом, має бути **скінченним** і тривати скінченний час. Окрім того, процес **не повинен обриватися** без отримання результату.

Приклад. Ділення числа в стовпчик, тобто утворення десяткового дробу, може бути залежно від конкретного числа скінченним або нескінченним (для $5/4$ дріб скінченний, для $5/3$ — ні). Якщо за деяким алгоритмом потрібно ділити числа, і дільником є 0, то ділення неможливе, і незрозуміло, що робити далі. ◀

Гарно побудований алгоритм має враховувати можливість нескінченного виконання дій або неможливість виконати наступну дію й містити команди, які забезпечують закінчення дій з деяким результатом **за будь-яких умов**.

У деяких ситуаціях розглядають **реальну виконуваність**, враховуючи вимоги до тривалості процесу, які висуваються в конкретних задачах. Існують такі задачі, для яких секунда — це занадто довго, а є й такі, що розв’язуються протягом кількох діб.

1.4 Мови програмування

Під **мовами програмування** розуміють штучні мови, призначені для запису програм. Першими мовами програмування були машинні мови, які вважають **мовами низького рівня**, адже дії комп’ютера представлені в них дуже деталізовано.

Машинна мова — це система команд, які може виконувати процесор. Кожна машинна команда записується послідовністю 0 та 1 і містить код машинної операції та адреси даних, до яких застосовується операція.

Програмування машинними мовами вимагає глибоких знань апаратної частини комп’ютера. Цими мовами користуються розробники комп’ютерів та деякі інші спеціалісти.

Для людини вони дуже незручні, тому в 1950-х роках було розпочато винайдення мов програмування, які за формою мали бути ближчими до мови математичних формул та людських мов, а також вільними від «машинних» подробиць. Водночас, ці мови мали бути такими, щоб записані ними програми можна було перекласти у відповідні машинні програми за допомогою самої машини. І такі мови програмування невдовзі було розроблено.

Дії комп’ютера в цих мовах було представлено з високим рівнем абстракції, тому ці мови стали називати **мовами високого рівня**. Комп’ютер безпосередньо не здатен виконувати

програми, написані мовами високого рівня. Такі програми необхідно обробляти **мовними процесорами** (інакше, **трансляторами**), наприклад, транслювати (перекладати) в програми на машинній мові (цей переклад виконує **компілятор**) або **інтерпретувати** за допомогою спеціальної програми — **інтерпретатора**, який зазвичай покомандно (або рядками) аналізує й відразу ж виконує програму, що подається йому на вхід. Як правило, інтерпретація вхідної програми відбувається повільніше, ніж виконання відповідної машинної програми. При цьому вхідна програма може попередньо перетворюватися в програму на проміжній мові, яка надалі виконується інтерпретатором.

1.5 Від тексту програми до її виконання

Для того, щоб написати програму мовою високого рівня та запустити її на виконання треба виконати кілька проміжних кроків. Ці кроки можна виконувати окремо та/або вручну або використовувати допоміжні програмні засоби, що спрощують роботу програміста.

Інтегроване середовище розробки (IDE, англ. *Integrated Development Environment*) — система програмних засобів, що використовується для розробки програмного забезпечення. Найтипівіше середовище зв'язує між собою текстовий редактор, компілятор та/або інтерпретатор та налагоджувач (англ. *debugger*) і тим самим надає програмісту зручний та наочний інтерфейс, що підвищує ефективність роботи програміста. Також середовище може надавати доступ до додаткових засобів, наприклад, засобів автоматизації контролю версій, засобів автоматичної перевірки стилю, засобів реорганізації (тобто рефакторингу) коду тощо. Для компільованих мов до середовища додають засоби автоматизації компонування. Для інтерпретованих мов можуть включатися засоби перевірки синтаксичної правильності тексту програми.

Розглянемо можливі кроки роботи з програмою.

Набирання тексту програми. Технологічні процеси, які дозволяють від тексту програми перейти до її виконання, у випадку компільованих та інтерпретованих мов дещо різняться, але в обох випадках текст програми відповідною мовою програмування спочатку слід набрати та зберегти в одному чи кількох файлах.

Текст програми мовою високого рівня (вхідний текст) найчастіше набирають за допомогою спеціальної програми (**текстового редактора**) і, як правило, записують на диск у вигляді **вхідного файлу** (англ. *source file*). Текст програми може складатися з кількох файлів — у великих програмах їх можуть бути десятки й сотні.

Компіляція. Цей крок притаманний переважно так званим компільованим мовам, для яких існують транслятори на машинну мову, але може виконуватися й для інтерпретованих мов. Під час компіляції перевіряється синтаксична правильність вхідного тексту та створюється відповідний **об'єктний код**, що є еквівалентом вхідного тексту на мові, дуже наближений до машинної. Для деяких інтерпретованих мов також може виконуватися компіляція, у результаті якої вхідний текст перетворюється в програму на проміжній низькорівневій мові (так званий **байт-код**), яка далі буде оброблятися інтерпретатором. Побудова байт-коду дозволяє дещо пришвидшити інтерпретацію програми.

Навіть за відсутності компілятора для інтерпретованих мов можуть існувати засоби перевірки вхідного тексту на синтаксичну правильність, які дозволяють здійснити перевірку без запуску програми на виконання. Доволі часто текстовий редактор поєднують з засобами компіляції та перевірки вхідного тексту, щоб відразу за набирання тексту давати програмісту підказки.

Компонування, або збирання, виконується для компільованих мов. Побудований компілятором об'єктний код обробляє ще одна програма — **компонувальник**. Ця програма «збирає з частин» (**компоує**) виконуваний код, тобто машинну програму, і, як правило, записує його на диск у вигляді файлу, **готового до виконання**. Такий файл можна завантажити пізніше.

Існують інтерпретовані мови програмування, які «імітують» компонування, а саме: вони записують в один виконуваний файл і код інтерпретатора, і байт-код, який має ним інтерпретуватися.

Запуск на виконання. У випадку виконання машинної програми вона спочатку записується (не обов'язково повністю) в оперативну пам'ять, тобто **завантажується**. Це здійснює спеціальна програма — **завантажувач**. Якщо завантаження здійснено успішно, починається процес **виконання** завантаженої програми. У випадку інтерпретованих мов в оперативну пам'ять спочатку має бути завантажений інтерпретатор, який далі обробляє вхідний текст програми на мові програмування (або байт-код).

Зрозуміло, що програма машинною мовою буде виконуватися швидше ніж байт-код, а байт-код буде виконуватися швидше, ніж безпосередня інтерпретація вхідного тексту. Наприклад тому, що у випадку машинної програми деякі попередні кроки обробки тексту було зроблено один раз та заздалегідь, а під час інтерпретації вхідного тексту вони виконуються за кожного виконання. З цієї точки зору компільовані мови мають однозначну перевагу над інтерпретованими.

Для запуску на виконання програми інтерпретованою мовою на комп'ютері має бути встановлений відповідний інтерпретатор. Як зазначалося раніше, деякі мови виходять з ситуації за допомогою поєднання інтерпретатора та байт-коду в одному виконуваному файлі. Але цей підхід не є масовим.

Для розповсюдження програми, записаної машинною мовою, достатньо передати саму програму (виконуваний файл) та, можливо, кілька системних бібліотечних файлів компілятора (наприклад, кілька dll-файлів з кодом системних бібліотечних функцій мови програмування), але це дуже мала частина від тих засобів, що входять до складу компілятора. Для розповсюдження програми, записаної інтерпретованою мовою, необхідно забезпечити наявність інтерпретатора. Очевидно, що з цієї точки зору перевага знов-таки за компільованими мовами.

Але поглянемо дещо інакше. Сьогодні існують різні операційні системи (ОС), у тому числі й різні за архітектурою, що впливає на набір службових підпрограм. Програма машиною мовою призначена для виконання під певними ОС може виявитися несумісною з іншими ОС, наприклад, через іншу розрядність або через використовуваний набір службових підпрограм ОС. Тому переносимість програм машинною мовою між різними ОС бажає кращого. Щоб мати можливість виконати програму, написану компільованою мовою, під певною ОС, треба побудувати виконуваний файл, що відповідає цій ОС. Щодо інтерпретованих мов, то тут простіше. Оскільки на комп'ютері, призначеному для виконання програми, має бути встановлений інтерпретатор, то саме цей інтерпретатор має відповідати ОС. Отже, з так званою кросплатформеністю в інтерпретованих мов справи значно кращі. І це явна перевага інтерпретованих мов.

Тим не менш слід зазначити, що деякі аспекти навіть інтерпретованих мов програмування можуть не переноситися між різними платформами, наприклад, якщо призначені для роботи саме із специфічними нюансами певної ОС.

Тестування та наладка. Не кожна програма починає працювати як задумано з першої спроби. Можливі помилки. «На пальцях» тестування полягає в перевірці правильності програми. У залежності від того, які аспекти програми перевіряються, воно може полягати в простому запуску програми на виконання з наступним порівнянням отриманих та очікуваних результатів та/або використовувати допоміжні програмні засоби, зокрема, інтегровані в середовище.

За виявлених проблем програмісту може знадобитися налагоджувач. **Налагоджувач** переважно використовується для локалізації помилок у тексті програми та в деяких випадках для тестування. Він забезпечує виконання вхідної програми невеликими порціями (кроками) і дає можливість побачити результати виконання після кожного кроку. Це полегшує виявлення помилок у вхідній програмі.

1.6 IDE для мови Python

Мова Python є інтерпретованою. Для виконання програм, написаних мовою Python, на комп'ютері має бути встановлений інтерпретатор мови Python.

На початку буде достатньо встановити Python, який вільно розповсюджується з офіційного сайту, див. <https://www.python.org/downloads/>. Слід вибрати відповідний інсталяційний пакет.

Важливо! Якщо збираєтесь встановити Python на комп'ютер, яким користується ще хтось інший і він вже встановив собі Python, то пам'ятайте, що сумісне використання одного облікового запису користувача може призвести до того, що наявність ще однієї, нової, версії може зіпсувати працездатність існуючих програм, якщо цю версію невдало встановити. Питання встановлення нової версії варто обговорити з власником існуючої інсталяції, щоб не зламати йому налаштування. Також з особливою обережністю та уважністю слід встановлювати програми від імені адміністратора.

Важливо для користувачів Windows! Якщо на комп'ютері до цього не було жодних версій Python, то під час інсталяції варто поставити галочку біля «Add Python 3.X to PATH». Якщо на комп'ютері до того навмисно чи випадково були встановлені інші версії Python, то інсталятор може запропонувати модифікувати існуючу версію. Можлива ситуація, коли інсталятор буде ставити ще одну версію Python. У цьому випадку галочку краще не ставити, а натомість потім вручну перевірити, яка саме версія згадується першою в системній змінній PATH та в змінній PATH користувача, під яким ви працюєте. Змінювати шлях до поточної версії Python у змінних PATH варто тільки тоді, коли є впевненість на 120%, що ці дії не викличуть проблеми з використанням раніше встановлених версій Python і особливо його сторонніх бібліотек.

Важливо для користувачів unix-комп'ютерів! Не перекривайте системну версію Python! Дійте згідно інструкції з встановлення на офіційному сайті Python <https://docs.python.org/3/using/unix.html>, розділ 2.2. Building Python, але в кінці завжди використовуйте не ~~make install~~, а `make altinstall`

Разом з інтерпретатором у випадку Windows буде встановлена документація та оболонка (англ. *shell*) IDLE, якої для того, щоб розпочати вивчення мови Python, цілком достатньо. Після встановлення Python можна встановити більш розвинуті IDE, наприклад:

PyCharm звідси <https://www.jetbrains.com/pycharm/download>.

У науковців доволі популярним є дистрибутив Anaconda, що складається з інтерпретатора мови Python та різноманітних бібліотек.

1.7 З історії створення мови Python

Мова Python виникла на початку 1990-х років у Нідерландах із аматорського проєкту однієї людини — Гвідо ван Россум (Guido van Rossum). Свою назву вона отримала на честь головного персонажа популярного в ті роки телевізійного серіалу «Monty Python's Flying Circus».

Одним з принципів мови є те, що текст програм має бути зрозумілим не гірше за англійську мову. Така «інтуїтивність» дозволяє зменшити зусилля непрограмістів для її використання. Завдяки численним бібліотекам, зокрема, й математичним, мова Python зараз доволі широко розповсюджена серед науковців. Ще одна широка галузь застосування мови Python — програмування для web та сервісне обслуговування (навіть у розробці програмного забезпечення іншими мовами програмування).

На сьогодні це доволі поширена мова, яку часто позиціонують як мову, яку легко вчити, використовувати, розуміти та використовувати. Також ця мова є кросплатформенною.

Серед явних недоліків мови є те, що написані на ній програми є доволі повільними. Тому для низькорівневого програмування її все-таки не використовують.

Існують різні реалізації мови Python. Найбільш поширеною є CPython — реалізація, що написана мовою C. Хоча існують й інші. Подальше стосуватиметься саме CPython.

Мова Python постійно розвивається та оновлюється. Поточною основною версією є Python 3, підтримка використовуваної раніше версії Python 2 припинена 01.01.2020. Станом на вересень 2025 року нова версія виходить вже не кожні півтора роки, а кожен рік. Зазвичай, під час оновлення програмних продуктів намагаються зберегти *обернену сумісність*.

Для прикладної програми під оберненою сумісністю розуміють можливість нової версії програми успішно працювати з даними старої версії. Для мов програмування під оберненою сумісністю розуміють те, що програма, написана під стару версію мови, не втрачає та не змінює свій зміст за переходу на нову версію мови. Нажаль, мова Python і її бібліотеки є достатньо нестабільними, особливо ті, що стосуються асинхронного програмування, і для мови Python обернена сумісність інколи може втрачатися.

Текст цього посібника орієнтується переважно на версію 3.14 мови Python.

1.8 Офіційна документація на мову Python

Офіційна документація на мову Python наведена на офіційному сайті мови і доступна за посиланням <https://docs.python.org/>.

Також для мови Python існують так звані «пропозиції з покращення» (Python Enhancement Proposals, або PEP), які розташовані за посиланням <https://www.python.org/dev/peps/>. Не всі вони втілені в мову, тобто мають статус документації (на них посилається офіційна документація). Деякі, наприклад, є відхиленими, деякі — тільки розглядаються, деякі є дуже рекомендованими для дотримання спільнотою програмістів мовою Python.

Одним з найбільш згадуваних документів є, наприклад, **PEP8 — Style Guide for Python Code**, який містить рекомендації щодо оформлення тексту програми. Дотримання цих рекомендацій є дуже бажаним, хоча, звісно, існують ситуації, коли їх не дотримуються і це недотримання має рацію. Деталі **PEP8** див. на офіційному сайті.

1.9 Перша програма — «Hello, World!»

Консольні програми мовою Python прийнято записувати у текстовий файл з розширенням **.py**, наприклад, **hello.py**.

Приклад. У книжках з мов програмування дуже часто першою наводять програму, яка «вітає світ», тобто виводить на екран рядок-привітання.

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

hello.py

Програма складається з єдиної інструкції — *виклику* функції **print**, — що виводить на консоль відповідне текстове повідомлення.

Запустити цю програму на виконання можна з командного інтерпретатора, виконавши рядок `python hello.py`

Примітка. Для цього шляхи до відповідного виконуваного файлу `python.exe` мають бути прописані в поточному значенні системної змінної `PATH`. Інакше слід вказувати повний шлях до `python.exe` та розширення.

Для Linux за цією командою за умовчанням може бути використаний інтерпретатор версії **Python 2**, тому під Linux відповідний рядок виклику повинен мати вигляд (також для запуску файлу з поточного каталогу Linux вимагатиме записати шлях до нього з `./`)

```
python3 ./hello.py
```

або так (якщо встановлено кілька версій Python)

```
python3.14 ./hello.py
```

Також в командному інтерпретаторі під Windows (якщо встановлено утиліту Python launcher for Windows) можна просто виконати рядок
`py hello.py`

Використовувана для інтерпретації версія Python також може регулюватися першим рядком файлу з текстом програми (у документації див. *shebang line*). Щоб такий спосіб запуску (за іменем файлу з текстом програми) спрацював під Linux, *shebang*-рядок просто необхідний.

Якщо запускати цю програму не з-під командного інтерпретатора, то можна помітити, що консольне вікно на мить з'являється та відразу зникає. Модифікуємо.

```
print("Hello, World!")  
input("Press ENTER to continue")
```

hello.py

За другою інструкцією буде виведено відповідне повідомлення та програма чекатиме натискання клавіші ENTER, після чого завершить виконання.

Примітка. Насправді **input** чекатиме на введення від користувача, яке має завершуватися натисканням клавіші **ENTER**.

Наведена програма нетипово коротка.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: парадигма програмування, декларативне програмування, імперативне програмування, комп'ютерна програма, програмування, модель, мова, синтаксис, семантика, штучна мова, природна мова, мова програмування, алгоритм, процес, детермінованість, екземпляр задачі, реальна виконуваність алгоритму, машинна мова, мова високого рівня, мовний процесор, транслятор, компілятор, інтерпретатор, інтегроване середовище розробки, IDE, завантажувач, байт-код, тестування, наладка, налагоджувач, обернена сумісність, PEP.

2. Які парадигми програмування ви знаєте?

3. До якої парадигми програмування відноситься мова Python?

4. Які бувають дані в алгоритмах та програмах?

5. Назвіть властивості алгоритму.

6. Про який рівень йдеться в терміні «мова високого рівня»?

7. Чим компіляція відрізняється від інтерпретації? Які переваги та недоліки мають ці підходи?

Чим вони обумовлені?

8. Що є входом та виходом компілятора та компоувальника в компільованих мовах?

9. Що є входом інтерпретатора? У чому полягає результат роботи інтерпретатора?

10. Які засоби зазвичай входять до складу типового інтегрованого середовища програмування?

11. Яку мову програмування вивчаємо (з точністю до назви та першої цифри версії)? ☺

12. У чому важливість дотримання оберненої сумісності? Чи завжди вона виконується для мови Python?

13. Яка реалізація мови Python розглядається в цьому посібнику?

14. Чи всі PEP є діючими документами, обов'язковими до дотримання?

15. Про що йдеться в PEP8?

16. Назвіть випадки, коли недотримання PEP8 має рацію.

2 Зображення інформації у комп'ютері

Розглянемо загальні принципи зображення інформації у комп'ютері, що не залежать від мов програмування.

2.1 Позиційні системи числення

2.1.1 Поняття системи числення

Система числення — це система правил запису чисел.

Людина звикла до десяткової системи числення, яка з'явилася в давніх Вавилоні та Індії, потім стала відома арабам і завдяки їм прийшла в Європу. У цій системі є *десять знаків* — цифр, якими позначають числа від **0** до **9**. Більші числа позначають *послідовностями* цифр. Знаки у послідовності займають різні **позиції (розряди)**: цифра праворуч позначає кількість одиниць, наступна — кількість десятків тощо. Цифра залежно від позиції має «різну вагу» (у записі **32** цифра **2** задає дві одиниці, а у **23** — два десятки). Цю систему запису чисел називають **позиційною**. Кількість знаків (цифр) у системі числення називається її **основою**.

Люди користуються записом чисел з основою не тільки 10, але й, наприклад, 12, 60 або 5. У програмуванні можна зустріти запис чисел з основою 8 та 16.

У десятковій системі цифра праворуч у запису числа називається **молодшою**, ліворуч — **старшою**. Розряд одиниць називається нульовим, десятків — першим, сотень — другим тощо. Розряд вказує, на який степінь числа 10 треба помножити цифру в цьому розряді (одиниці — на 10^0 , десятки — на 10^1 , сотні — на 10^2 тощо). Отже, число записується як сума добутків цифр числа на відповідні степені десятки, наприклад, $5834 = 5 \cdot 10^3 + 8 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$.

Якщо запис числа має дробову частину, то додаються цифри, ділені на 10 у відповідних степенях, наприклад, $0,234 = 2 \cdot \frac{1}{10^1} + 3 \cdot \frac{1}{10^2} + 4 \cdot \frac{1}{10^3}$.

Так само числа записуються й з іншими основами більше 1. Питання лише, які знаки є цифрами. Якщо основа не більше 10, беруть звичні десяткові цифри (стільки знаків, скільки треба). Проте за основи більше 10 потрібні додаткові знаки, щоб позначати десяткові числа 10, 11, ... У XX столітті для цього стали використовувати послідовні великі літери латинського алфавіту — A, B, C тощо¹.

Систему числення з цифрами від 0 до $P-1$ (де $P > 1$) називають **P-ковою**. P-ковий запис $x_n \dots x_1 x_0, x_{-1} \dots x_{-k}$, де всі $x_i \in P$ -ковими цифрами, є **скінченим** P-ковим зображенням числа

$$(x_n \dots x_1 x_0, x_{-1} \dots x_{-k})_P = x_n P^n + \dots + x_1 P^1 + x_0 P^0 + x_{-1} P^{-1} + \dots + x_{-k} P^{-k}$$

Запис $x_n \dots x_1 x_0, x_{-1} \dots x_{-k} \dots$ з P-ковими цифрами $x_i \in P$ є **нескінченим** P-ковим зображенням числа

$$(x_n \dots x_1 x_0, x_{-1} \dots x_{-k} \dots)_P = x_n P^n + \dots + x_1 P^1 + x_0 P^0 + x_{-1} P^{-1} + \dots + x_{-k} P^{-k} + \dots$$
$$= \sum_{i=0}^n x_i P^i + \sum_{i=1}^{+\infty} x_{-i} P^{-i}$$

Приклади

1. У двійковій системі числення лише дві цифри — **0** та **1**. Двійкові записи **10**, **11**, **100**, **101** позначають десяткові 2, 3, 4, 5. Запис **1101** позначає число $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$, запис **0,101** — число $1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0,625$.

2. У системі числення з основою 8 для запису чисел є цифри **0, 1, 2, 3, 4, 5, 6, 7**, «вага» кожного розряду числа є відповідним степенем вісімки. Отже, перші числа записуються як **0, 1, ... 6, 7**, а

¹ Також використовують й малі латинські a, b, c тощо.

далі йдуть записи **10, 11, ..., 17, 20, 21, ..., 77, 100, ...** Вісімкове **10** — це звичне десяткове 8, **11** — звичне 9, **20** — десяткове 16, тобто двічі по 8. А вісімкові записи **100, 200** та **400** — це десяткові 64, 128 та 256, тобто один, два та чотири рази по 8 у квадраті. Так само,

$$342_8 = 3 \cdot 8^2 + 4 \cdot 8^1 + 2 \cdot 8^0,$$

$$0,342_8 = 3 \cdot 8^{-1} + 4 \cdot 8^{-2} + 2 \cdot 8^{-3},$$

$$34,2_8 = 3 \cdot 8^1 + 4 \cdot 8^0 + 2 \cdot 8^{-1}$$

(про вісімковий запис свідчить маленька цифра 8 внизу).

3. У шістнадцятковій системі числа від 10 до 15 позначають так: 10 — **A**, 11 — **B**, 12 — **C**, 13 — **D**, 14 — **E**, 15 — **F**. Тоді $2A_{16} = 2 \cdot 16^1 + 10 \cdot 16^0 = 42$, $FF_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255$, $0,C_{16} = 12 \cdot 16^{-1} = \frac{3}{4} = 0,75$. ◀

2.1.2 Перетворення чисел з недесятькової системи в десяткову

У комп'ютері числа зображуються у двійковій системі числення. Проте людині зручніше вводити числа в комп'ютер та отримувати їх від нього як десяткові. Виникає **задача перетворення запису числа** з однієї системи в іншу (зокрема, з двійкової в десяткову й навпаки).

Запис $x_n \dots x_1 x_0, x_{-1} \dots x_{-k}$ з P -ковими цифрами x_i задає число $x_n P^n + \dots + x_1 P^1 + x_0 P^0 + x_{-1} P^{-1} + \dots + x_{-k} P^{-k}$. Цифри x_i у цій сумі позначають числа від 0 до $P-1$. Звідси, щоб знайти десяткове зображення числа, потрібно:

записати в десятковій системі число P та P -кові цифри числа;

обчислити суму добутків значень цифр

та відповідних степенів числа P .

Приклади

$$10011_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19;$$

$$10,011_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 2,375;$$

$$1BC_{16} = 1 \cdot 16^2 + 11 \cdot 16^1 + 12 \cdot 16^0 = 444;$$

$$1B,C_{16} = 1 \cdot 16^1 + 11 \cdot 16^0 + 12 \cdot 16^{-1} = 27,75;$$

$12,2_3 = 1 \cdot 3^1 + 2 \cdot 3^0 + 2 \cdot 3^{-1} = 5,666 \dots$, тобто число записується нескінченним періодичним десятковим дробом. ◀

2.1.3 Перетворення цілих чисел з десяткової системи в недесятькову

Розглянемо, як за натуральним числом N у десятковому записі отримати цифри P -кового зображення. Невідомі ані самі цифри, ані їх кількість. Проте відомо, що P -ковий запис задає число як суму добутків значень цифр на відповідні степені числа P , тобто за деякого невідомого n і невідомих цифр x_i справджується рівність

$$N = (x_n \dots x_1 x_0)_P = x_n P^n + \dots + x_1 P^1 + x_0 P^0$$

Помітимо: всі доданки, окрім останнього, мають множник P . Тоді значенням молодшої цифри x_0 є остача від ділення N на основу P , а сума $T = x_n P^{n-1} + \dots + x_1 P^0 = (x_n \dots x_1)_P$ дорівнює цілій частці від ділення N на P . Поділивши цю суму на P з остачею, знайдемо остачу x_1 і наступну частку, і так далі, поки на якомусь кроці частка від ділення не стане рівною 0. Остання остача й буде старшою цифрою x_n .

Приклади. Утворимо вісімковий запис числа **202**.

$$202 : 8 = 25 \text{ (остача } 2 \text{ — молодша цифра),}$$

$$25 : 8 = 3 \text{ (остача } 1 \text{ — наступна цифра).}$$

$$3 : 8 = 0 \text{ (остача } 3 \text{ — остання, старша цифра).}$$

Отже, отримано цифри **2, 1, 3** (від молодшої до старшої), тобто вісімковий запис **312**.

Запишемо 202 у 16-овій системі.

$$202 : 16 = 12 \text{ (остача } 10 \text{ — значення молодшої цифри } A),$$

$$12 : 16 = 0 \text{ (остача } 12 \text{ — значення старшої цифри } C),$$

Отже, десяткове 202 має 16-овий запис **CA**. ◀

Позначимо остачу від ділення T на P націло як $T \% P$, частку — як T / P . Опишемо утворення P -кового запису числа N таким алгоритмом.

1. Спочатку частка T дорівнює N , а запис порожній.
2. Поки $T > 0$,
 - обчислити остачу R як $T \% P$ та нову частку T як T / P (шляхом ділення у стовпчик),
 - зобразити R у P -ковій системі числення як цифру C , відповідну R ,
 - та дописати цифру C до запису ліворуч.

2.1.4 Перетворення дробових чисел з десятикової системи в недесяткову

Розглянемо, як за додатним дійсним числом $V < 1$ отримати цифри його P -кового запису (принаймні, кілька перших, оскільки запис може бути нескінченним). За способом запису,

$$V = (0, x_{-1} \dots x_{-k} \dots)_P = x_{-1}P^{-1} + \dots + x_{-k}P^{-k} + \dots$$

з невідомими значеннями цифр. Помножимо обидві частини рівності на P й отримаємо рівність

$$VP = x_{-1} + x_{-2}P^{-1} \dots + x_{-k}P^{-k+1} + \dots = (x_{-1}, x_{-2} \dots x_{-k} \dots)_P = x_{-1} + (0, x_{-2} \dots x_{-k} \dots)_P$$

Звідси $x_{-1} = [VP]$, а $(0, x_{-2} \dots x_{-k} \dots)_P = \{VP\}$, де $[VP]$ та $\{VP\}$ позначають цілу та дробову частини VP . Далі так само помножимо $\{VP\}$ на P , знову отримаємо цілу та дробову частину (ціла буде значенням x_{-2}), і так далі.

Приклади. Утворимо двійковий запис десятикового дробу 0,75.

$$0,75 \cdot 2 = 1,5, [1,5] = 1 \text{ (перша цифра)}, \{1,5\} = 0,5;$$

$$0,5 \cdot 2 = 1, [1] = 1 \text{ (наступна цифра)}, \{1\} = 0.$$

Усі подальші цифри будуть **0**, тому **0,11** є скінченним двійковим зображенням для десятикового 0,75.

Утворимо двійковий запис десятикового дробу 0,1.

$$0,1 \cdot 2 = 0,2, [0,2] = 0 \text{ (перша цифра)}, \{0,2\} = 0,2;$$

$$0,2 \cdot 2 = 0,4, [0,4] = 0 \text{ (наступна цифра)}, \{0,4\} = 0,4;$$

$$0,4 \cdot 2 = 0,8, [0,8] = 0 \text{ (наступна цифра)}, \{0,8\} = 0,8;$$

$$0,8 \cdot 2 = 1,6, [1,6] = 1 \text{ (наступна цифра)}, \{1,6\} = 0,6;$$

$$0,6 \cdot 2 = 1,2, [1,2] = 1 \text{ (наступна цифра)}, \{1,2\} = 0,2.$$

Далі цифри **0, 0, 1, 1** будуть повторюватися до нескінченності, тобто точний двійковий запис десятикового 0,1 є періодичним **0,0(0011)**. Будь-який початок цього запису позначає наближення до десятикового 0,1, наприклад, $0,00011_2 = \frac{1}{16} + \frac{1}{32} = 0,09375$ (помилка — приблизно шість тисячних).

Утворимо 16-ковий запис десятикового дробу 0,8.

$$0,8 \cdot 16 = 12,8, [12,8] = 12 \text{ (перша цифра } C), \{12,8\} = 0,8.$$

Далі цифра **C** буде нескінченно повторюватися, тому **0,(C)** є точним 16-ковим записом десятикового 0,8. ◀

- Якщо V є скінченним десятиковим дробом і основа P має обидва прості множники 2 та 5, то число V можна зобразити скінченним P -ковим записом. Інакше P -ковий запис може бути нескінченним; тоді скінченна кількість кроків дає *наближене зображення* числа V .

Отже, за дійсним числом V , $V < 1$, можна одержати R перших цифр його P -кового зображення, виконуючи такий алгоритм.

1. Спочатку зображенням є «0.».
2. Поки одержано менше за R дробових цифр,
 - обчислити VP , обчислити d як $[VP]$ (ціле число від 0 до $P-1$) та V як $\{VP\}$.

Записати значення d як P -кову цифру та дописати її до зображення праворуч.

Цілі та дробові числа перекладаються з однієї системи числення в іншу за *різними алгоритмами*. Тому, перекладаючи число, що має цілу та дробову частини, треба спочатку виділити ці частини й потім перекласти їх *окремо*.

Узагалі, щоб перекласти запис числа з довільної системи числення в іншу, необхідно виконати багато арифметичних дій у незвичній недесятковій системі. Інколи простіше скористатися десятковою системою як проміжною.

2.1.5 Знаходження дробових чисел за їхніми періодичними зображеннями

Розглянемо метод на прикладі. Раніше ми знайшли, що $\frac{1}{10} = (0,0(0011))_2$. Спробуємо знайти значення $(0,0(0011))_2$, не перекладаючи $\frac{1}{10}$ у двійкову систему числення. Нехай $x = (0,(0011))_2$, тоді $(0,0(0011))_2 = x \cdot 2^{-1}$.

Пам'ятаємо, що ділення на основу системи числення призводить до переносу десяткової крапки на одну позицію праворуч, а множення — ліворуч.

Помножимо x на 2^4 .

$$2^4 \cdot x = ((0,(0011))_2) 2^4 = (00011,(0011))_2 = (11,(0011))_2 = (11)_2 + (0,(0011))_2 = 3 + x.$$

Звідси маємо рівняння $16x = 3 + x$, з якого $x = \frac{1}{5}$.

Отже, $(0,0(0011))_2 = \frac{1}{10}$.

2.2 Зображення цілих чисел

Усі дані в комп'ютері зображуються, або *кодуються*, у вигляді послідовностей **0** та **1**. Зображення чисел за допомогою **0** та **1** називається *двійковим*, тобто зображенням у двійковій системі числення.

Стандартне зображення чисел у комп'ютері займає кілька послідовних байтів. Один байт може мати $2^8 = 256$ станів і зображати кожним з них одне з 256 значень, наприклад, ціле число від 0 до 255, або ціле число від -128 до 127 (яких теж 256), або щось інше. Два байти можуть мати $2^{16} = 65536$ станів і зображати ними 2^{16} різних значень тощо.

Цілі числа зображуються в комп'ютері переважно у двох формах — без знаку та зі знаком. Ці форми називаються *кодами*. Числа ототожнюються з їх зображеннями, хоча з погляду математики це є хибним.

Беззнаковий код займає кілька байтів (найчастіше, 1, 2, 4 або 8). Один байт зображує числа від 0 до 255 у їх двійковому записі, можливо, з незначущими нулями. А саме, байт **00000000** — число 0, байт **00000001** — число 1, байт **00000010** — число 2 тощо. Байти **1111 1110** та **1111 1111** зображають «останні» числа $254 = 2^8 - 2$ і $255 = 2^8 - 1$.

Додавання 1 до числа з кодом **1111 1111** дає послідовність бітів **1 0000 0000**, тобто $255 + 1 = 256$. Перенос 1 зі старшого розряду збільшує кількість цифр і не дозволяє зобразити суму 256 в одному байті. Перенос 1 зі старшого розряду називається *переповненням* (англ. *overflow*). Якщо ігнорувати переповнення, то про одnobайтові числа можна сказати ось що:

— «наступним» за найбільшим числом є найменше;

— якщо сума двох одnobайтових чисел більше 255, то їх одnobайтова сума на 256 менше справжньої суми.

За більшої кількості байтів, якими зображено числа, ситуація аналогічна, лише діапазон чисел інший. Так, два байти зображують числа від 0 до 65535 (це $2^{16}-1$), чотири — від 0 до 4294967295 (це $2^{32}-1$) тощо.

Знакові коди найчастіше займають 1, 2, 4 або 8 байтів. Старший біт зображує знак числа: **0** — «+», **1** — «-». Додатні числа зображуються так само, як і беззнакові, лише за рахунок знакового біта їх діапазон — від 0 до $2^{8N-1} - 1$, де N — кількість байтів. Таке зображення називається **прямим кодом**. Наприклад, число $2^{8N-1} - 1$ має прямий код **011...1**. Від'ємні числа мають так званий **додатковий код** (англ. *two's complement*). Код від'ємного числа A позначається $D(A)$ й утворюється так.

1. За прямим кодом числа $|A|$ заміною всіх 0 на 1 і всіх 1 на 0 отримуємо **обернений код** (англ. *one's complement*) $R(A)$.

2. За $R(A)$ як беззнаковим цілим числом обчислюємо код $D(A) = R(A) + 1$.

Приклад. Побудуємо однобайтовий додатковий код числа **-72**. Почнемо з того, що знайдемо двійковий запис числа $72 = 64 + 8 = 2^6 + 2^3 = 1001000_2$. Подальші кроки зображено в табл. 1.

Табл. 1 — Побудова додаткового коду числа **-72**

Прямий код $ A $	0100 1000
Обернений код $R(A)$	1011 0111
Додавання 1	1
Додатковий код $D(A)$	1011 1000

«Відновити» за додатковим кодом від'ємне число можна у зворотному порядку.

1. $D(A)$ розглядаємо як беззнакове ціле, обчислюємо $R(A) = D(A) - 1$.

2. Код, обернений до $R(A)$, є прямим кодом числа $|A|$.

Отже, однобайтові коди від **0000 0000** до **0111 1111** зображують числа від 0 до 127, а коди від **1000 0000** до **1111 1111** — числа від **-128** до **-1**. Аналогічно беззнаковим кодам, додавання 1 до 127 дає перенос 1 у знаковий розряд, тобто, ігноруючи перенос, отримуємо код числа **-128**. Так само, якщо сума двох додатних однобайтових чисел більше 127, то в одному байті залишається зображення від'ємного числа, яке на 256 менше справжньої суми.

У двох байтах зі знаком зображуються числа від **-32768** до **32767**, тобто від -2^{15} до $2^{15} - 1$, у чотирьох — від **-2147483648** (це -2^{31}) до **2147483647** (це $2^{31} - 1$) тощо.

2.3 Зображення дійсних чисел

2.3.1 Формати запису дійсних чисел

Розглянемо приклад. Число **-12,34** можна зобразити по-різному. По-перше, його можна записати звичайним чином у десятковій системі числення у вигляді **-12,34**, де кома (роздільник цілих та дробових розрядів) безпосередньо розділяє цілі та дробові розряди, тобто розташована на фіксованій позиції. Про такий запис можна сказати, що число зображено у **форматі з фіксованою крапкою**² (англ. *fixed*).

Нормалізоване (або, більш точно, **нормалізоване експоненційне**) зображення цього ж числа має вигляд **-1,234·10¹**, і складається з трьох частин: знаку (для додатних чисел знак може не записуватися), мантиси й порядку. Число **1,234** є **мантисою**, або **значущою частиною** (англ. *mantissa, significand*) нашого числа, а число **1** — **порядком** (англ. *exponent*). У нормалізованому зображенні ненульового дійсного числа мантиса за абсолютною величиною має належати проміжку [1; 10). У програмуванні про число, зображене в нормалізованому вигляді, кажуть, що воно записано в **«науковому»** (англ. *scientific*) форматі.

² У національній системі запису чисел роздільником цілих та дробових розрядів є кома, але в мовах програмування використовується крапка, тому все-таки «з фіксованою крапкою».

2.3.2 Зображення дійсних чисел у комп'ютері

Як бачили раніше, одне те саме число можна записати по-різному, навіть, якщо воно має скінченне зображення. У той же час, деякі числа взагалі не мають скінченного зображення й їх можна записати тільки наближено.

Множина дійсних чисел нескінченна та всюди щільна (між кожними двома різними дійсними числами знайдеться ще нескінченно багато). Довільне кодування може точно зобразити лише *скінченну підмножину*, решта може бути зображена тільки *наближено*.

Якщо взяти за основу зображення числа *з фіксованою крапкою*, то можна домовитися зберігати числа із фіксованою кількістю знаків після коми. Це може бути корисним, наприклад, коли дійсне число позначає кількість грошових одиниць. Навряд чи при неправильному нарахуванні податків інспектора влаштує пояснення, що число **0,1** не можна точно зобразити у двійковій системі числення й тому за певний період накопичилася арифметична помилка. Такого вигляду зображення дійсних *не є звичайними для процесора*, тому далі **не розглядаються** (хоча в мові Python наявний тип **decimal**)³.

Противагою до них є зображення *з плаваючою крапкою* (англ. *floating-point*), які ґрунтуються на ідеї нормалізованого зображення. Бажано, щоб результати обчислень над дійсними числами не залежали від того, на якому процесорі виконується програма, на якій мові вона була написана і яким компілятором компілювалася. З цією метою у 1985 році було розроблено стандарт арифметики з плаваючою крапкою, який у 2008, а згодом у 2019 році було дещо перероблено. Сьогодні обробка чисел з плаваючою крапкою в комп'ютерних системах (як програмних, так й апаратних) регламентується стандартом *IEEE std. 754 - 2019 (Standard for Floating-Point Arithmetic)*, якого мають дотримуватися виробники комп'ютерних систем (як апаратних, так й програмних).

Надалі розглянемо тільки деякі принципи кодування дійсних чисел, а не ретельно весь стандарт. Процесор (точніше, модуль операцій з плаваючою крапкою) обробляє дійсні числа, що кодуються **0** та **1**. Тому розглянемо принципи будови двійкових кодувань. Стандарт розглядає двійкові кодування дійсних у 4, 8 та 16 байтах. Тут розглянемо 8-байтове кодування дійсних чисел, тобто в 64 бітах: саме воно переважно використовується в сучасних процесорах (і, як наслідок, в більшості мов програмування).

У двійковій системі числення також можна казати про нормалізовані зображення дійсних чисел. Так, у двійковій системі мантиса має належати проміжку $[1; 2)$. Розглянемо нормалізоване двійкове зображення числа $(-1000,110011)_2$:

$$(-1000,110011)_2 = -(1,000110011)_2 \cdot 2^3.$$

За стандартом 64 біти розподіляються для кодування знаку, порядку та мантиси. У першому біті кодується знак, наступні $w=11$ біт кодують порядок, решта, $t=52$ біти, кодують мантису.

Кодування знаку. **1** в полі знаку кодує «мінус», а **0** — «плюс». У нашому прикладі відповідний біт приймає значення **1**.

Кодування порядку. Порядок може приймати як додатні, так й від'ємні значення, тому в полі порядку насправді записується 11-бітовий беззнаковий код зсунутого порядку (*biased exponent*) E , який більший за порядок на величину зсуву $bias = 1023$.

У нашому прикладі порядок дорівнює 3, тому зсунутий порядок E дорівнює

$$E = 3 + 1023 = 1026 = 1024 + 2 = (10000000010)_2.$$

Відповідна послідовність бітів є такою **10000000010**.

Кодування мантиси. Оскільки ціла частина мантиси завжди дорівнює 1, то достатньо записати перші 52 двійкових розряди її дробової частини (на рис. *trailing significand field*).

Для нашого прикладу це буде

³ У деяких мовах програмування та базах даних такий підхід втілено в реалізацію типів типу **decimal**.

Хоча й нуль має два зображення, але в порівняннях знак має ігноруватися, тобто, $-0 = +0$.

Також вважається, що $-\infty < \text{довільне скінченне число} < +\infty$.

Невизначеності було введено для того, щоб за довільних аргументів операції над дійсними числами могли що-небудь повернути. Так, наприклад, значенням виразів $0 \cdot \infty$, ∞/∞ , $0/0$, $\sqrt{x}$, де $x < 0$, у **процесорі** є невизначеність. Результати обчислення відповідних виразів (а також тих, значеннями яких є нескінченності) у конкретній мові програмування можуть бути іншими (через спробу транслятора попереджати та/або обробляти помилки)!

Згідно стандарту значення NaN не дорівнює самому собі (тобто $\text{NaN} \neq \text{NaN}$ є істинним).

З наведеного випливає.

- Дійсні числа в описаному зображенні точно зображують тільки скінченну підмножину розширеної множини дійсних чисел. Деякі числа точно зобразити неможна.
- Між двома послідовними числами в описаному зображенні є **ненульова різниця**. Вона дуже мала, якщо числа близькі до 0, і досягає десятків тисяч, якщо числа близькі до найбільшого за модулем. Зазвичай, у мовах програмування для програміста доступні такі значення, як найбільше та найменше скінченне дійсне, найменше додатне дійсне, відстань від 1 до найближчого більшого.
- Результат обчислення арифметичних операцій над дійсними числами може не співпадати з математичним (за рахунок округлень).
- Додавання та множення дійсних чисел не є асоціативними (за рахунок різних округлень, які можуть виникати під час обчислень).

2.4 Зображення символів

Символи (англ. *character*) в комп'ютері також кодуються послідовностями 0 та 1.

Історично першим в інформатиці було кодування символів у 7 бітах (бо використовувалося для передачі символів по каналах зв'язку) і кодувало символи цілими числами від 0 до 127 ($2^8 - 1$). Відповідність між числами від 0 до 127 та символами зафіксовано в **Американському стандартному коді обміну інформацією** (так звана **таблиця ASCII**).

Однією з властивостей таблиці ASCII є таке:

- символам 0, 1, ..., 9 відповідають послідовні коди від 48 до 57;
- символам A, B, ..., Z — від 65 до 90;
- символам a, b, ..., z — від 97 до 122;
- символ «пробіл» має код 32.

Згодом почали використовувати 8-бітове кодування (в одному байті), додавши символи з кодами від 128 до 255. Цей байт як зображення числа без знаку дає **код символу** (число від 0 до 255). Відповідність між символами та кодами встановлюється таблицями.

Отримана кодова таблиця з кодами від 0 до 255 доволі часто згадується як «розширена ASCII». Нижня частина кодової таблиці (коди від 0 до 127) є кодуванням US-ASCII і визначається стандартом. Щодо кодів від 128 до 255, то для них єдиного стандарту немає. Серед них можна виділити різноманітні кодування, що включають символи національних алфавітів, наприклад, KOI-8. Для кирилических символів, крім модифікацій KOI-8, існують ще кілька різних кодових таблиць, що згадуються в ОС сім'ї Windows під номерами 1251 (Windows-1251) та 866 (DOS).

Сьогодні крім однобайтових зображень широкого вжитку набули багатобайтові кодування, які краще пристосовані до національних особливостей та забезпечують кращу переносимість текстової інформації, бо насправді не залежать від національної специфіки.

Інший поширений стандарт кодування символів — Unicode. Він складається з універсального набору символів (англ. *Universal character set*, UCS), що задає символи та їхні коди, та сімейства кодувань (англ. *Unicode transformation format*, UTF), що задають зображення цих кодів. Також

універсальний набір символів задає належність символів певним категоріям, наприклад: цифри; великі, малі, ... літери (символи національних алфавітів); порожні (або білі) символи (англ. *spaces, white symbols, whites*) тощо.

Так звані **порожні**, або **білі**, **символи** не мають зображення. Під час набирання тексту вони створюються за натискання клавіш **Space** (пробіл), **Enter** («кінець рядка») і **Tab** (символ табуляції)⁴. Насправді, така категорія символів існує не тільки для Unicode, але й для інших кодувань. Зазначені символи відносяться до білих не тільки в Unicode, але й за використання кодування ASCII.

У стандарті Unicode коди від 0 до 127 відповідають тим самим символам, що й за кодування **ASCII**.

UTF-8 для кодування символів може використовувати від 1 до 4 байтів. Символи US-ASCII у кодуванні UTF-8 зображуються 1 байтом, більш того вміст цього байту такий самий, як за використання кодування **ASCII**.

Кодування UTF-32 зображує символи Unicode в 4 байтах, кодування UTF-16 використовує 2 байти на кожен символ (за винятком деяких символів, що кодуються так званою сурогатною парою, тобто два рази по 2 байти).

Для кодування символу двома байтами виникає питання: який байт записано першим: старший чи молодший. Якщо спочатку записується молодший, то таке кодування називають *little-endian* (LE); якщо спочатку старший, то це *big-endian* (BE) кодування. Так отримуються кодування UTF-16LE та UTF-16BE. За порядком розташування пар байтів аналогічно отримуються кодування UTF-32LE та UTF-32BE.

Щоб позначити, що текстовий файл записано універсальним набором символів та для визначення кодування, на початку файлу можуть записувати **маркер послідовності байтів** (*byte order mark, BOM*). Порядок байтів для UTF-8 фіксований, але BOM все одно може мати місце, щоб позначити, що файл записано в кодуванні UTF-8. Зауважимо, що за стандартом для файлу в кодуванні UTF-8 BOM не зазначається. Маркери різних кодувань наведено в табл. 2.

Табл. 2 — Маркери різних кодувань

Байти на початку файлу	Кодування
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8

У Windows кодуванню UTF-8 відповідає кодова таблиця 65001.

Текстові редактори здатні використовувати різні кодування. Менш сучасні редактори, особливо під кирилізованою Windows, доволі часто використовують кодування 1251. При цьому в них є засоби конвертації до іншого кодування, у тому числі в різні UTF.

Для текстів програм сучасні IDE все найчастіше за умовчанням використовують кодування UTF-8. І це є ознакою гарного тону використовувати саме кодування UTF-8, яке сприяє роботі з кодом інтернаціональних команд розробників (на відміну від одnobайтових кодувань) і при цьому доволі економне за використанням пам'яті (символи US-ASCII кодуються в ньому в одному байті).

⁴ Тут наведено не вичерпний перелік білих символів.

Щодо кодування тексту програми **мовою Python**, то за умовчанням воно розглядається як **UTF-8**.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: система числення, позиційна система числення, основа системи числення, молодша цифра/розряд у запису цілого числа, старша цифра/розряд у запису цілого числа, цифри 16-ї системи числення, період, беззнаковий код, прямий код, обернений код, додатковий код, переповнення розрядів, формат запису з фіксованою крапкою, нормалізований формат запису, мантиса, значуща частина, порядок, денормалізовані дійсні числа, NaN, ASCII, Unicode, UTF, UTF-8, UTF-16, UTF-32, BOM, порожній (білий) символ.

2. Як отримується значення цілого числа, записаного в системі числення з основою P ?

3. Сформулюйте алгоритм, який дозволяє записати ціле число в системі числення з основою P ?

4. Як отримується значення дійсного числа, записаного в системі числення з основою P ?

5. Чи кожне дійсне число має скінченний запис в системі числення з основою P ?

6. Сформулюйте алгоритм, який дозволяє знайти перші K дробових знаків запису дійсного числа в системі числення з основою P ?

7. Сформулюйте алгоритм, який дозволяє записати раціональне число в системі числення з цілою основою P ($P \geq 2$)?

8. Дійсне число має періодичний запис у системі числення з основою P . Сформулюйте алгоритм, який дозволяє знайти його значення.

9. Чи кожне дійсне число можна точно зобразити в комп'ютері у 8 байтах? А в 1024 байтах?

10. Чи використовуються в програмуванні формати зображення дійсних чисел з фіксованою крапкою?

11. На якій ідеї ґрунтуються формати зображення дійсних чисел з плаваючою крапкою?

12. Який формат зображення дійсних чисел використовується в сучасних процесорах?

13. Опишіть, як отримати 8-байтове зображення дійсного числа у форматі з плаваючою крапкою.

14. Чи доречно за використання 8-байтового зображення казати про «сусідні» дійсні числа? Як змінюється відстань між «сусідами» за збільшення абсолютної величини числа?

15. Скільки для дійсного числа може існувати 8-байтових зображень у форматі з плаваючою крапкою?

16. Які дійсні числа мають декілька 8-байтових зображень у форматі з плаваючою крапкою?

17. Чи кожне 8-байтове зображення у форматі з плаваючою крапкою задає деяке дійсне (з точки зору математики) число? Якщо не всі, то вказати які.

18. Як виконуються порівняння «особливих» дійсних чисел (+0, -0, $+\infty$, $-\infty$, NaN) між собою та зі звичайними дійсними числами.

19. В яких випадках може бути корисно мати як +0, так і -0?

20. Чи можна число π точно зобразити у форматі з плаваючою точкою? Якщо так, то скільки байтів для цього потрібно?

21. Чи є математична функція $\lg$ всюди визначеною на «комп'ютерній» множині звичайних дійсних чисел? Які можливі проблеми з її обчисленням?

22. Чи може добуток двох ненульових «комп'ютерних» дійсних чисел дорівнювати нулю?

23. Чи завжди сума (різниця, добуток, частка) двох скінченних «комп'ютерних» дійсних чисел є а) дійсним числом; б) «комп'ютерним» дійсним числом?

24. Сума (різниця, добуток, частка) деяких двох «комп'ютерних» дійсних чисел є скінченним «комп'ютерним» дійсним числом. Чи завжди це число дорівнює математичному значенню суми (різниці, добутку, частки)?

25. Чи є додавання та множення «комп'ютерних» дійсних чисел у форматі з плаваючою крапкою асоціативним?

26. У результаті обчислень було отримано два дійсних значення. Як можна перевірити їх на рівність? Які можливі потенційні проблеми?

27. Вкажіть особливості кодування цифр та латинських літер у таблиці ASCII.

28. Чи розрізняються коди арабських цифр та малих та великих латинських літер таблиці US-ASCII та Unicode?

29. Яке з кодувань UTF-8, UTF-16, UTF-32 буде найбільш економним для тексту, що використовує тільки символи з таблиці US-ASCII? Відповідь обґрунтувати.

30. Для чого використовується BOM?

31. В якому кодуванні прийнято записувати текст програм мовою Python?

Задачі та вправи

1. Перекласти запис 10111 з системи числення з основою $P = 2, 8, 16$ до системи числення з основою 10.

2. Перекласти записи FF, 4AB з 16-ї системи до 10-ї.

3. Записати число 10111 у системах числення з основами $P = 2, 8, 16$.

4. Знайти двобайтовий та чотирибайтовий додатковий код чисел 1, 8, 9, 255, 256, 640, 641, -1, -8, -9, -255, -256, -640, -641, -32767, -32768.

5. Як змінюється додатковий код цілого числа за збільшення розрядності?

6. Записати власну дату народження у форматі **dd.mm.yyyy** у системах числення з основами $P = 2, 8, 16$.

7. Записати $(10,011)_2$, $(0,10(01))_2$ у десятковій системі числення.

8. Записати $1/2$, $1/3$, $2/7$, $1/10$ у системах числення з основами $P = 2, 8, 16$.

9. Які числа мають скінченний запис у системі числення з цілою основою P ($P \geq 2$)? Охарактеризуйте відповідну множину чисел.

10. Які числа мають періодичний запис у позиційній системі числення з цілою основою?

11. Скільки різних записів у системі числення з цілою основою P ($P \geq 2$) може мати дійсне число? Які числа мають більше одного запису? Охарактеризуйте відповідну множину.

12. Записати 13.125, 1024, 127.125, $1/2$, $1/3$, $1/10$ у 8-байтовому кодуванні з плаваючою крапкою.

13. Знайти найменше додатне дійсне число в 8-байтовому кодуванні. (Денормалізовані числа не розглядати.)

14. Знайти найменше додатне денормалізоване дійсне число в 8-байтовому кодуванні.

15. Знайти найбільше скінченне дійсне число в 8-байтовому кодуванні.

16. За використання 8-байтового зображення знайти найближче до числа 1 дійсне, що його перевищує. Чому дорівнює відстань між ними?

17. За використання 8-байтового зображення знайти скінченне дійсне число, яке є «сусідом» найбільшого скінченного дійсного, та відстань між ними.

18. Чи правда, що за стандартом $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1$ дорівнює 1.0 ?

19. Чи правда, що за стандартом $0.5+0.5+0.5+0.5+0.5+0.5+0.5+0.5+0.5$ дорівнює 5.0 ?

20. Чи можуть дві різних 8-байтових зображення кодувати одне те саме дійсне число?

21. Знайти три таких дійсних значення x , y , z , що для їх 8-байтових зображень з фіксованою крапкою $(x + y) + z \neq x + (y + z)$.

3 Типи даних, арифметичні дії, вирази

Комп'ютер може обробляти дані різної природи. Так, рік народження людини ми зображаємо цілим числом, вага може бути не лише цілою, але й дробовою, а прізвище позначаємо послідовністю символів. Якщо числа можна додавати та віднімати, то послідовності символів можна дописувати одна до одної. Отже, різні за своєю природою дані (значення) мають різні способи зображення та допустимі операції, і за цими ознаками поділяються на типи.

Перед тим, як розглядати типи даних, що підтримуються мовою Python, розглянемо поняття типу даних та які вони бувають узагалі, якщо не зосереджуватися на конкретній мові програмування.

3.1 Поняття типу даних в інформатиці

Тип даних — це множина значень разом із множиною застосовних до них операцій. Множина значень називається **носієм** типу, множина операцій — **сигнатурою**.

У багатьох мовах програмування використовуються типи чисел (цілих та з плаваючою точкою), символічних та логічних значень (істина/хибність). Ці значення розглядаються як цілісні елементи, що не мають окремих складових частин, тому називаються **скалярними**. Типи скалярних даних називаються **скалярними**.

Противагою скалярним значенням є **складені** (англ. *composite*), що складаються з окремих компонентів. Складені типи даних ще можуть називати **складними**, рідше **структурованими**. Прикладом складених значень є точка площини, що моделюється парою координат: складається з окремих компонентів.

У програмуванні до складених типів відносять, зокрема, масиви, структури, а також класи. (Хоча, якщо значення типу, що є класом, не має окремих складових частин, то такий клас іноді можуть відносити до скалярних типів.) Класи є втіленням поняття типу даних у тому розумінні, що насправді об'єднують у собі не тільки значення, але й означення застосовних до них операцій. Усі перетворення значення, чий тип є класом, мають визначатися виключно через означені в типі операції. Це дозволяє унеможливити зміни даних у недопустимий спосіб, захистити дані від несанкціонованих змін, і тим самим підвищити надійність коду.

Усі типи мови Python є класами.

Кожна мова програмування забезпечує низку вбудованих типів, які доступні програмісту відразу та без жодних додаткових дій. Також програмісти можуть використовувати типи з стандартної бібліотеки (містить набір засобів, що є невід'ємною частиною мови програмування), з сторонніх бібліотек (розроблених іншими програмістами) та створювати власні типи, зокрема, власні складені типи та означати операції над ними.

3.2 Вбудовані числові типи даних мови Python

Мова Python має кілька вбудованих (доступні відразу, без жодних додаткових дій програміста) типів, для зображення чисел:

- тип **int** для зображення цілих чисел;
- тип **float** для зображення дійсних чисел;
- тип **complex** для зображення комплексних чисел.

Крім них у стандартній бібліотеці означено типи **Decimal** та **Fraction**. Вбудований логічний тип **bool** формально теж числовий, оскільки є похідним від типу **int**.

Розглянемо вбудовані типи більш детально.

3.2.1 Тип цілих чисел *int*

Як зазначалося раніше, цілі числа зображуються значеннями типу **int**. Згідно з принципом абстракції даних деталі внутрішнього зображення документацією не розкриваються, натомість постулюються певні властивості даних цього типу.

Якщо дуже коротко, то *принцип абстракції даних* зводиться приблизно до такого: щоб використати певну річ, достатньо знати як нею можна користуватися; знати у всіх подробицях, як вона влаштована всередині, при цьому не обов'язково. Цей принцип застосовується не тільки до програмних систем, але оточує нас у повсякденному бутті. Кожного дня ми користуємось побутовими пристроями, різними гаджетами, тим самим комп'ютером, як «чорними скриньками», тобто не знаючи подробиць того, як все влаштовано всередині. Натомість до кожного пристрою виробником додається інструкція користувача, де сказано, як ним користуватись.

На відміну від багатьох інших мов, в яких цілі числа зображуються скалярним типом, значення якого займають фіксовану кількість байтів, а тому можуть моделювати тільки скінченний та задалегідь обмежений діапазон цілих чисел, діапазон цілих чисел мови Python, що можуть бути зображені в типі **int**, є потенційно необмеженим (насправді, єдиним обмеженням буде кількість доступної віртуальної пам'яті комп'ютера). При цьому саме ціле значення розглядається ззовні як неподільне.

У тексті програми (та в рядку інтерпретатора) цілі значення можна записувати в десятковій, шістнадцятковій, вісімковій та двійковій системах числення, вони можуть починатися зі знаку (+ або -).

Так, кожен із десятих записів **0**, **12**, **-14**, **150** задає відповідне ціле число. Запис, що починається з двох або більшої кількості нулів поспіль, наприклад, **0012**, на відміну від деяких інших мов програмування у тексті програми є помилковим.

Записи **0b10**, **0B11**, **-0b11** сприймаються як двійкові і задають цілі числа **2**, **3** та **-3** відповідно. Ознакою того, що число записано у двійковій системі числення є префікси **0b** та **0B**.

Записи **0o10**, **0O11**, **-0O11** (тут **o**, **O** — мала та великі латинські літери 'o', від *octal* — вісімковий) сприймаються як вісімкові і задають цілі числа **8**, **9** та **-9** відповідно. Ознакою того, що число записано у вісімковій системі числення є префікси **0o** та **0O**.

Ознакою шістнадцяткової системи числення є префікси **0x** та **0X**. В якості шістнадцяткових цифр використовуються цифри від 0 до 9 включно та літери **a**, **b**, **c**, **d**, **e**, **f**, **A**, **B**, **C**, **D**, **E**, **F**, при цьому малі та великі відповідні «цифри» мають однакове значення. Так, кожен із записів **0xff**, **0xFF**, **0Xff** задає ціле **255**.

3.2.2 Тип дійсних чисел *float*

Як зазначає документація на CPython, дійсні числа типу **float** моделюються з використанням типу **double** мови C. Станом на поточний момент це означає, що для самого значення використовується 8-байтове зображення дійсних з плаваючою крапкою згідно із стандартом IEEE 754. (Тобто використовується те саме зображення, що його використовують виробники процесорів.)⁵

З способу зображення випливає, що:

- 1) тільки скінченна підмножина множини дійсних чисел може бути зображена в типі **float**;
- 2) множина дійсних значень типу **float** не є щільною; більш того, інтервали між «сусідніми» дійсними числами зростають із зростанням абсолютної величини.

Дійсні числа записуються в тексті програми виключно в десятковій системі числення. Можливе використання, як формату з *фіксованою fixed*) *крапкою*, так і *нормалізованої*

⁵ Точну інформацію щодо поточної реалізації дійсних чисел в інтерпретаторі під час виконання можна отримати від класу **sys.float_info**.

(«*наукової*», англ. *scientific*) форми запису, наприклад, записи **12.34**, **1234.0e-02**, **1234e-02**, **0.1234E2**, **0.01234E+3** позначають одне й те саме значення типу **float**, що наближає число **12,34**. Роздільником цілої та дробової частини є крапка, а не кома.

Ознакою того, що запис зображує дійсне число є наявність крапки та/або наявність порядку.

За використання формату з фіксованою крапкою обов'язково має бути крапка та хоча б одна десяткова цифра. Наприклад, **3.14**, **314.**, **0.314**, **.314**, **.1**, **00012.34**. З останнього прикладу видно: на відміну від записів цілих значень, записи дійсних значень можуть починатися з кількох нулів поспіль. Якщо ціла частина відсутня, вона вважається рівною **0**. Записи **0.314** та **.314** позначають одне те саме дійсне значення. Запис **314** позначає значення цілого типу, у той же час запис **314.** позначає значення дійсного типу.

Записи дійсних значень у нормалізованій формі починаються з запису дійсного значення або непорожньої послідовності десяткових цифр, за якими слідує порядок. Порядок починається з літери **e** або **E**, за якою може йти знак, тобто **+** або **-** (якщо знак відсутній, то вважається **+**), і далі непорожня послідовність десяткових цифр. Наприклад, **12.3e+02**, **.3E+2**, **12.3e02**, **012.3e02**, **12.3E-02**, **02e-10**.

Отже, дійсні значення:

- завжди записуються в 10-й системі числення;
- роздільником цілої та дробової частини є крапка;
- до їх складу обов'язково входить дійсна крапка або порядок;
- порядок обов'язковим не є;
- за наявності дробової частини нульова ціла частина може опускатися.

Також у типі **float** наявні значення **+∞**, **-∞** та **NaN**. У тексті програми вони спеціальних позначень не мають, натомість можуть бути отримані, наприклад, з використанням бібліотечних засобів.

3.2.3 Тип комплексних чисел *complex*

Серед вбудованих числових типів у мові Python присутній тип комплексних чисел **complex**. Як відомо, у математиці комплексне число має дійсну та уявну частину. Якщо в математиці уявна частина позначається суфіксом **i**, наприклад, **10+2i**, то в мові Python використовуються суфікси **j** та **J**. Наприклад, записи **10.0+2.5j**, **13J**, **0.5j** позначають комплексні числа типу **complex**.

3.2.4 Логічний тип *bool*

Логічний тип **bool** (від англ. *boolean*) є похідним⁶ від типу **int** та має тільки два можливих значення: «хибність» та «істина», що в тексті програм (та в інтерпретаторі) позначаються як **False** та **True** відповідно. Їх називають *булевими* на честь видатного англійського логіка й математика Джорджа Буля.

Логічні значення, що позначаються як **False** та **True**, відповідають цілим **0** та **1**.

3.3 Прості арифметичні дії над числовими даними

У мові Python над числовими даними означено додавання, віднімання, множення, ділення та піднесення в степінь, які позначаються як **+**, **-**, *****, **/** та ****** відповідно.

Для всіх розглянутих типів, крім **complex**, означено операції цілочислового ділення (ділення націло) та взяття остачі від цілочислового ділення, що позначаються **//** та **%** відповідно. Значення, що отримується в результаті цілочислового ділення **x** на **y**, дорівнює **[x/y]**, де **[]**

⁶ Неформально можна сказати, що похідний тип є уточненням типу, від якого є похідним.

позначає цілу частину (англ. *floor*) — найбільше ціле число, що не перевищує задане. Наприклад, $[1] = 1$, $[1.1] = 1$, $[1.5] = 1$, $[1.9999] = 1$, $[-1.1] = -2$, $[-1.5] = -2$, $[-1.9999] = -2$.

По аналогії з математикою значення $(x//y)*y + (x\%y)$ має дорівнювати x . Якщо остача ненульова, то її знак співпадає зі знаком дільника. На відміну від математики цілочисельне ділення та взяття остачі застосовне як до цілих, так і до дійсних значень. У випадку дійсних аргументів через неточність зображення дійсних чисел в окремих випадках значення $x//y$ може виявитися на 1 менше математичного. Значення $(x//y)*y + (x\%y)$ є доволі близьким до x .

Значеннями виразів $3//2$, $(-3)//2$, $3//(-2)$, $(-3)//(-2)$ є цілі 1 , -2 , -2 , 1 відповідно.

Значеннями виразів $3\%2$, $(-3)\%2$, $3\%(-2)$, $(-3)\%(-2)$ є цілі 1 , 1 , -1 , -1 відповідно.

Значення виразу $1.5//0.75$ дорівнює дійсному 2.0 .

Так само, як і в математиці, ділити на 0 , брати остачу від ділення на 0 та зводити 0 у від'ємний степінь, не можна. Натомість, значенням 0^0 є 1 .

У табл. 3 вказано тип результату арифметичних операцій за умови застосування до аргументів *одного того самого числового типу*.

Табл. 3. — Арифметичні операції у мові Python

Тип аргументів	Операція	Тип результату
bool	$+$, $-$, $*$, $//$, $\%$, $**$	int <i>Примітка.</i> Під час обчислень False розглядається як 0 , True — як 1 .
int	$+$, $-$, $*$, $//$, $\%$	int
int	$**$ та показник невід'ємний	int
int	$**$ та показник від'ємний	float
float	$+$, $-$, $*$, $//$, $\%$, $**$	float
bool, int, float	$/$	float
complex	$+$, $-$, $*$, $/$, $**$ <i>Примітка.</i> Операції $//$ та $\%$ незастосовні.	complex

Примітка. Крім наведених вище операцій до цілих значень у мові Python застосовні зсуви (англ. *shift*) і так звані **побітові** (англ. *bitwise*) операції: операція виконується не над значенням у цілому, а над відповідними бітами його зображення. З огляду на те, що внутрішнє зображення цілих у мові Python не є відкритим, вважається, що побітові операції виконуються так, *наскільки аргументи записано в додатковому коді*.

Розглядають $<<$ та $>>$ — лівий та правий зсув, за яких біти лівого аргументу зсуваються на вказану у другому аргументі кількість позицій ліворуч або праворуч відповідно. За лівого зсуву молодші біти заповнюються нулями.

Побітові операції позначаються такими операторами:

$\sim$ — інверсія, унарна операція за якої значення кожного біту змінюється на протилежне (тобто 0 на 1 , 1 на 0); вирази $\sim n$ та $-(n+1)$ задають одне те саме значення;

$\&$ — побітове "і";

$\wedge$ — побітове "виключне або";

$|$ — побітове "або".

Детально оператори $<<$, $>>$, $\sim$, $\&$, $\wedge$, $|$ над числовими значеннями в межах цього посібника не розглядатимуться.

3.4 Обчислення в інтерпретаторі

Інтерпретатор Python можна використовувати як калькулятор. Він дозволяє інтерактивно виконувати інструкції: набрана інструкція за натискання **Enter** відразу виконується. Якщо набрати якийсь вираз, то він буде обчислений і його значення буде відразу виведено.

Для завантаження інтерпретатора Python можна, наприклад, у рядку командного інтерпретатора запустити на виконання виконати **python** (або **python3**, або **python3.14**, ... у залежності від операційної системи та встановленої версії Python, детальніше див. підрозділ 1.9). Дещо інший варіант отримати доступ до інтерпретатора — запустити, наприклад, IDLE.

У табл. 4 наведено послідовність введених інструкцій та відповідь на них інтерпретатора.

Табл. 4 — Арифметичні обчислення в інтерпретаторі

Вікно інтерпретатора	Примітки
>>> 0b0100	Ціле у двійковому запису перетворено у внутрішнє зображення, отримане ціле значення виведено в десятковому запису.
4	
>>> 1+2	
3	
>>> 1/2	
0.5	Результат обчислення виразу є дійсним, тому 2.0, а не 2.
>>> 2/1	
2.0	Число 0.1 зображується в дійсному типі наближено, тому сума десяти 0.1 не дорівнює 1.0, звідси маємо ненульову різницю.
>>> 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1-1.0	
-1.1102230246251565e-16	За використовуваного в цьому експерименті процесора та інтерпретатора ця різниця виявилася нулевою. Щаслива випадковість. Можливо в когось вона не буде нулем. Але в цілому тут жодної фантастики нема.
>>> 0.1+0.1+0.1+0.1+0.1-0.5	
0.0	

Примітка. Найчастіше в очікуванні введення інтерпретатор виводить запрошення у вигляді >>>. Набирати ці символи власноруч не треба.

Дізнатися тип значення можна за допомогою **type**. Значенням виразу вигляду **type(expr)** є тип виразу **expr**. За спроби його вивести відповідне значення конвертується в рядок, який би був інформативним для людини. Розглянемо приклад визначення типів в інтерпретаторі (табл. 5).

Табл. 5 — Визначення типів виразів за допомогою інтерпретатора

Вікно інтерпретатора	Примітки
>>> type(12)	У Python всі типи є класами, що підтверджує вихід інтерпретатора.
<class 'int'>	
>>> type(2/1)	

<code><class 'float'></code>	Як бачимо, результат ділення є дійсним, навіть якщо два цілих націло діляться одне на друге.
------------------------------------	--

Якщо інтерпретатор було викликано з командного рядка, то за інструкцією **quit()** його робота буде завершена і керування повернеться командному інтерпретатору. Якщо інструкцію виконати з IDLE, то виконання IDLE буде завершено.

3.5 Вирази

3.5.1 Вирази та їх складові

Згадаємо математику. Припустимо, що необхідно знайти площу трикутника, для якого відома довжина однієї зі сторін та висоти, проведеної до цієї сторони. Як можна записати площу? Можливо, що так:

нехай

a – довжина сторони трикутника,

h – довжина висоти трикутника, проведеної до цієї сторони,

тоді площа трикутника дорівнює $(a \cdot h)/2$.

Що ми бачимо в цій ситуації. У нас є вираз $(a \cdot h)/2$, який задає обчислення площі трикутника та містить у собі імена **a**, **h**, позначення операцій $\cdot$, $/$, константу **2** та дужки.

Якщо дивитися більш уважно, то можна сказати, що імена **a**, **h** позначають деякі змінні величини. Поняття змінної числової величини з'явилося в роботах Декарта в XVII ст. Пізніше з'ясувалося, що змінна величина може бути не лише числовою. У найширшому розумінні **змінна величина** — це загальне поняття реального чи уявного об'єкта (або його окремої характеристики), який може перебувати в різних станах.

Обчислювальні дії у програмуванні називаються **операціями**. Операції застосовуються до **операндів**, тобто значень. Застосування операцій до значень описується у вигляді **виразу** (англ. *expression*). Послідовність застосування операцій називається **обчисленням** виразу й має результатом **значення виразу**. Операції (у найпростішому випадку) у виразі позначаються знаками (**операторами**)⁷, а значення — константами та іменами (змінних). У виразі також можуть бути дужки, які визначають порядок застосування операцій. Найпростішими виразами є вирази, що не містять операцій, тобто константи та імена змінних.

Приклад. Вираз **2+2** означає: додаються 2 й 2 і значенням виразу є 4; вираз **2*radius** — 2 множиться на значення змінної **radius** і значенням виразу є подвоєне значення цієї змінної; **1+2*3** — множаться 2 і 3, отриманий добуток 6 додається до 1 і значенням є 7; **(1+2)*3** — додаються 1 і 2, їх сума 3 множиться на 3 й значенням є 9. За двома останніми виразами видно, як дужки впливають на порядок операцій. Також виразами є **3** та **radius**. ◀

Вираз має подвійну семантику — задає **послідовність операцій** з операндами, а також **значення**, що є результатом цієї послідовності.

За кількістю операндів виділяють **двомісні**, або **бінарні** (мають два операнди), та **одномісні**, або **унарні** (мають один операнд) операції (та оператори); вони зустрічаються найчастіше. Операції з більшою кількістю операндів також можливі. Наприклад, у деяких мовах програмування існують й тернарні оператори (мають три операнди).

Бінарні операції ******, *****, **/**, **//**, **%**, **+**, **-**, застосовні до вбудованих числових типів, розглядалися раніше. Зараз звернемо увагу на те, що в мові Python існують унарні операції **+** та **-**. Для типів **int**, **float**, **complex** жодна з унарних операцій не змінює тип значення; унарний **+** не змінює значення, унарний **-** змінює знак на протилежний. Тип **bool** є похідним від **int**, обчислення

⁷ Також операції можуть позначатися іменами, наприклад, **sin(x+y)**.

унарних операцій для нього відбувається за тими ж правилами, що й для бінарних операцій; деталі наведено в табл. 6).

Табл. 6 — Унарні арифметичні операції мови Python

Тип аргументу	Унарна операція	Результат
bool	+	Результат має тип int . Значенням +False є 0 , значенням +True є 1 . <i>Примітка.</i> Під час обчислень False розглядається як 0 , True – як 1 .
bool	-	Результат має тип int . Значенням -False є 0 , значенням -True є -1 .
int, float, complex	+	Тип результату збігається з типом аргументу. Результат збігається зі значенням аргументу.
int, float, complex	-	Тип результату збігається з типом аргументу. Результат відрізняється від значення аргументу тільки знаком.

Наприклад, якщо значенням змінної **a** є **13**, то значенням виразів **+a** та **-a** є **13** та **-13** відповідно.

Зауваження. Математично вираз **-a** еквівалентний виразу **(-1)*a**, але унарний мінус (особливо для дійсних значень) обчислюється набагато швидше за множення на **-1**. Ця оптимізація доволі часто використовується в математичних обчисленнях.

3.5.2 Старшинство операторів та порядок виконання операцій

Мова Python, в основному, відповідає угодам математики про порядок застосування операцій у виразах. Це дозволяє не записувати зайві дужки, наприклад, **1-2*3** означає те ж, що й **1-(2*3)**. На порядок обчислення виразу за відсутності дужок впливає *старшинство* (*precedence*), або *пріоритет*, операторів: якщо поруч із позначенням операнду записано два оператори, то спочатку виконується операція, що відповідає старшому оператору (з вищим пріоритетом).

Окрім пріоритетів, оператори мають властивості право- або лівобічного зв'язування. У мові Python двомісні оператори *****, **/**, **%**, **//**, **+**, **-**, мають властивість *лівобічного зв'язування*⁸: якщо ліворуч і праворуч від позначення операнду записано знаки операцій з однаковим старшинством, то операнд зв'язується з оператором, вказаним ліворуч (ця операція застосовується спочатку). Інколи неформально кажуть, що операції обчислюються зліва направо. Наприклад, вираз **1-2-3** буде обчислюватися як **(1-2) -3**.

Оператор ****** має властивість *правобічного зв'язування*: вираз **2**2**3** буде обчислюватися справа наліво як **2**(2**3)**.

Оператори *****, **/**, **//**, **%** називають *мультиплікативними*, а бінарні оператори **+**, **-** — *адитивними*. По аналогії з математикою та багатьма іншими мовами програмування, пріоритет мультиплікативних операторів вищий за пріоритет адитивних. При цьому мультиплікативні оператори є рівнопріоритетними між собою. Так само рівнопріоритетними між собою є адитивні оператори.

У таблиці 7 наведено деякі оператори мови Python в порядку спадання пріоритетів. Оператори, розташовані в одній комірці є рівнопріоритетними.

⁸ Оператори, що мають властивість лівобічного зв'язування ще називають *лівоасоціативними*. *Правоасоціативні* оператори мають властивість правобічного зв'язування.

Табл. 7 — Пріоритети операторів мови Python

Оператор	Примітка
**	Має властивість правобічного зв'язування.
унарні +, -	
мультиплікативні *, /, //, %	Мають властивість лівобічного зв'язування
адитивні бінарні +, -	Мають властивість лівобічного зв'язування

Приклади

1. Значенням виразу $4 + 7//5$ є 5, оскільки спочатку обчислюється $7//5$ з результатом 1, а потім $4+1$ з результатом 5. Значенням виразу $(4+7)//5$ є 2, оскільки спочатку обчислюється операція у дужках $(4+7)$ — її значення 11, — а потім $11//5$ з результатом 2.

2. Значення виразу $4 - 3 - 2$ дорівнює -1, оскільки спочатку обчислюється $4-3$, тобто 1, а потім $1-2$; у виразі $2*7\%8$ спочатку обчислюється $2*7$ (це 14), потім $14\%8$, тобто 6.

3. За виразів з цілими, де вони діляться та множаться, порядок обчислень варто програмувати проводячи математичний аналіз. Особливо, якщо математичний результат є цілим числом. Так, $2*3//6$ дорівнюватиме 1, проте значенням $2//6*3$ буде 0.

4. Значенням виразу $2**2 + 2$ є 6, оскільки спочатку обчислюється $2**2$ (це 4), а потім $4+2$.

5. Пріоритет оператора ****** найвищий за пріоритет інших наведених у таблиці операторів, що є його «сусідами» зліва, навіть унарних, наприклад $-1**2$ буде обчислюватися як $-(1**2)$. У той же час, вираз $2**-1$ буде обчислюватися як $2**(-1)$.

◀

Пріоритети операторів дозволяють не записувати зайві дужки, але *зловживати цим не слід*. Інколи зайва пара дужок значно підвищує зрозумілість запису. Також зрозумілість запису підвищують пробіли: так пробіли навколо менш пріоритетного оператора у виразі $x*y + z*t$ спрощують його сприйняття навіть за відсутності явно записаних дужок. До речі, такий підхід до розстановки проміжків рекомендований у PEP8.

Зауваження. Хоча пріоритети та властивості ліво-/правобічного зв'язування задають порядок виконання операцій у складних виразах, але до кінця не фіксують порядок обчислення. Коли ми обчислюємо вираз вигляду $x*y + z*t$ виникає питання: який з підвиразів $x*y$ та $z*t$ обчислювати першим. Документація на мову Python дає відповідь на це питання: обчислення виразу відбувається зліва направо, тобто спочатку буде обчислено $x*y$, а потім $z*t$. За обчислення виразу $expr1 + expr2 * (expr3 - expr4)$ його підвирази $expr1$, $expr2$, $expr3$, $expr4$ будуть обчислюватися в порядку індексів.

3.6 Виконання арифметичних дій над числовими значеннями різних типів

З точки зору алгебри, арифметичні операції мають виконуватись над аргументами з однієї множини значень. Тим не менш, у програмуванні арифметичні дії можна виконувати не тільки над числовими значеннями одного того самого типу, але й над значеннями різних числових типів.

Сумісністю відмінних між собою типів у виразах називається можливість сумісного використання операндів цих типів. Типи **bool**, **int**, **float**, **complex** є сумісними. Під час

обчислення над різнотипними числовими значеннями спочатку виконується неявне перетворення типу одного з аргументів, а потім вже виконується обчислення.

Перетворення типу — утворення значення одного типу за значенням іншого типу. Також вживають термін **приведення до типу**. Перетворення типу вважається **неявним**, якщо виконується без явної вказівки програміста на його виконання. Для числових типів під час обчислення арифметичних операцій виконуються такі правила:

- якщо одне зі значень є комплексним, то інше неявно перетворюється на комплексне;
- інакше, якщо одне зі значень є дійсним, то інше неявно перетворюється на дійсне;
- інакше операція виконується над цілими значеннями (логічні значення є частковим випадком цілих, тому перетворення непотрібне, а під час обчислень **False** розглядається як **0**, **True** — як **1**).

Так, результатом обчислення **1+2.0** буде дійсне **3.0**; результатом обчислення **3**False** буде ціле **1**.

3.7 Явне перетворення типів

Перетворення типів значень, описані вище, відбуваються без явних на це вказівок, тобто неявно. Але перетворення можна задати **явно**. Для перетворення до типів **bool**, **int**, **float**, **complex** можна використовувати конструкції типу **позначення_типу(аргумент)**.

За перетворення на тип **bool** ненульові числові значення перетворюються на **True**, нульові перетворюються на **False**.

За перетворення на тип **int** **True** перетворюється на **1**, **False** на **0**. Щодо перетворення дійсних, то утворити ціле значення за дійсним можна або відкиданням дробових розрядів, або округленням. Щодо **CPython**, то там використовується такий самий механізм, як і в мові C: ціле значення утворюється за дійсним відкиданням дробових розрядів. Під час перетворення дійсних значень до цілого типу математично значення може змінитися.

Якщо ціле число може бути зображено в типі **float**, то перетворення до типу **float** зберігає математичне значення; інакше виникає помилка. Дійсні значення перетворюються до типу **complex** із збереженням математичного значення.

Комплексні числа до типів **int** та **float** не перетворюються, навіть за нулевої уявної частини.

Приклади

1. Значенням **bool(1)**, **bool(-13)**, **bool(13.6)**, **bool(1+1j)** є логічне **True**; значенням **bool(0)**, **bool(0.0)**, **bool(0+0j)** є **False**.

2. Значенням виразу **int(2.8)** є **2**, виразу **int(-2.8)** є **-2**.

3. Значенням виразу **float(2)** є дійсне **2.0**.

4. Значенням виразу **complex(2)** є **2+0j**. Зазначимо, що комплексні числа виводяться в дужках, отже в інтерпретаторі побачимо:

```
>>> complex(2)
(2+0j)
```

5. Значенням виразу **complex(1, 2)** є комплексне **1+2j**.

◄

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: тип даних, носій типу, сигнатура, скалярний тип даних, складений тип даних, вбудовані числові типи мови Python, абстракція даних, формат з фіксованою крапкою, нормалізована форма запису, значення логічного типу, арифметичні операції над числовими типами, сумісність типів, явне та неявне перетворення типу, операція, операнд, вираз, унарні оператори, бінарні оператори, пріоритет

операторів, лівобічне зв'язування, правобічне зв'язування, мультиплікативні оператори, адитивні оператори.

2. Як одним словом можна класифікувати усі типи мови Python?
3. В якому типі зображуються цілі числа в мові Python? Який у нього діапазон значень?
4. Які системи числення використовуються для запису цілих чисел у мові Python?
5. В якому типі зображуються дійсні числа в мові Python?
6. Які системи числення використовуються для запису дійсних чисел у мові Python?
7. Як називається логічний тип у мові Python? Як позначаються його значення?
8. Які арифметичні операції наявні у мові Python? До яких типів застосовні? Який тип має результат?
9. Застосування до значень арифметичних операцій мови Python є помилковим?
10. Які особливості має обчислення цілочислового ділення та частки?
11. Чи можна додавати логічні значення?
12. Як можна дізнатися тип результату обчислення виразу?
13. Як виконуються арифметичні операції над значеннями різних типів?
14. Як задати явне перетворення типу?
15. Що задає вираз у програмуванні?
16. Вказати арифметичні оператори мови Python у порядку спадання пріоритету?
17. Які оператори мають властивість лівобічного, а які властивість правобічного зв'язування?
18. У якому порядку обчислюються підвирази складного виразу в мові Python?
19. Чи можна програмувати математичну формулу буквально, як її записано? Чому?
20. Чи варто мінімізувати кількість дужок у виразі, покладаючись на пріоритети операторів?
21. Яких правил варто притримуватися при текстовому оформленні виразу?

Задачі та вправи

1. Використовуючи інтерпретатор мови Python, знайти число, яке позначає запис 10101, якщо основою системи числення є $P = 2, 8, 16$.
2. Використовуючи інтерпретатор мови Python, знайти значення $(FF)_{16}$, $(4AB)_{16}$, $(FAFBCD)_{16}$.
3. Указати значення виразів мови Python та їх тип:
 - а) $9/5$, $(-9)/5$, $9/(-5)$, $(-9)/(-5)$;
 - б) $9//5$, $(-9)//5$, $9//(-5)$, $(-9)//(-5)$;
 - в) $9\%5$, $(-9)\%5$, $9\%(-5)$, $(-9)\%(-5)$;
 - г) $9.0/5.0$, $9.0//5.0$, $1.5//0.75$.
4. Указати значення виразів мови Python та їх тип:
 - а) $7/3$, $1/6$, $7./3.$, $1./6.$;
 - б) $7//3$, $1//6$, $7.//3.$, $1.//6.0$.
5. Указати значення виразів мови Python та їх тип:
 - а) $1+4//2$, $(1+4)//2$;
 - б) $2*4\%7$, $2*(4\%7)$;
 - в) $51//6\%7$, $51/(6\%7)$;
 - г) $2*(7\%8)$, $12//6\%8$, $5-3\%2$.
6. Указати значення виразів мови Python та їх тип:
 - а) $4*6//8$, $4//8*6$;
 - б) $4*6/8$, $4/8*6$.
7. Указати значення виразів мови Python та їх тип:
 - а) $(-3+5)*(2\%7/3+4*2)$, $(-3+5)*(2\%7//3+4*2)$;
 - б) $(3/5)*20+32.0$, $(3//5)*20+32.0$, $(3//5.0)*20+32$, $(3//5.0)*20+32.0$.
8. Указати значення виразів мови Python та їх тип:
 $-2**4+1$, $2**-4+1$, $4**0.5$, $0**0$, $2**3*3$, $10**2$, $10**2.0$
9. Указати значення виразів мови Python та їх тип:

`True**True+True, 10**False, True*2+False`

10. Указати значення виразів мови Python та їх тип:

`int(4.1), int(-4.1), int(4.5), int(-4.5), int(4.7), int(-4.7),
float(14), complex(13), complex(1,1), int(bool(25)),
float(bool(1.5)), float(int(1.56)), bool(-0.0), int(bool(13.56))`

11. Нехай v позначає ціле невід'ємне число. Написати вираз, який задає обчислення:

а) значення молодшої десяткової цифри числа v ;

б) значення молодшої двійкової цифри числа v .

12. Записати вираз для обчислення значень

а) $\frac{10.001^{145} \cdot 13.001^{149}}{9.001^{155} \cdot 11.001^{179}}$; б) $\frac{10.001^{345} \cdot 13.001^{249}}{9.001^{355} \cdot 11.001^{269}}$; в) $\frac{20}{3} \cdot \frac{12}{5} \cdot \frac{11}{7}$; г) $\frac{12.11 \cdot 71}{3.5 \cdot 9}$.

13. Записати вирази для обчислення значень

а) $\sqrt{2}$; б) $3^{1.5}$; в) $(-2)^{25}$; г) $(-2)^{26}$.

14. Записати вираз, відповідний математичному виразу

а) $\sqrt{a^2 + b^2}$; б) $(a + b)^{1/7}$; в) $\sqrt[3]{a^{12} + b^{12}}$.

15. Відомо, що значення виразу $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1-1.0$ в інтерпретаторі відображається як $-1.1102230246251565e-16$, а значення виразу $0.1+0.1+0.1+0.1+0.1-0.5$ в інтерпретаторі дорівнює нулю. Як це можна пояснити?

16. За допомогою інтерпретатора знайти найменше ціле значення n , що $10^{-n} = 0.0$.

4 Лексичні елементи програми мовою Python

В обробці тексту програми транслятором (залежно від мови, інтерпретатором та/або компілятором) традиційно виділяють фази лексичного та синтаксичного аналізу (англ. *parsing*). Для деяких мов програмування фази лексичного аналізу може передувати попереднє оброблення тексту препроцесором (спеціальна програма, яка виконує певне перетворення вхідного тексту, наприклад, деякі службові підстановки тощо).

У природніх мовах найменшою одиницею, що має зміст, є слово. Для мов програмування аналогами слів природньої мови є лексеми.

- **Лексема** — послідовність символів вхідного тексту програми, яка розглядається транслятором як неподільна й самі по собі має деякий зміст.

Задача лексичного аналізатора полягає в тім, щоб розбити вхідний текст на лексеми. Отриманий потік лексем передається синтаксичному аналізатору (неформально кажучи, парсеру від англ. *parser*), однією з задач якого є перевірка синтаксичної коректності (якщо зовсім неформально, то правильності утворення речень зі слів).

У цьому розділі розглядатимуться лексеми мови Python.

4.1 Алфавіт та словник мови Python

У випадку мови Python входом лексичного аналізатора є текст програми мовою Python. За умовчанням вважається, що текст програми записано в кодуванні UTF-8 (деталі див. **PEP 3120**), хоча наявна можливість указати інше кодування вхідного текстового файлу. Якщо текст програми не може бути розкодованим, то виникає помилка⁹.

Будь-яка мова має **алфавіт** — скінченну множину символів, дозволених для використання. Алфавіт для написання програм мовою Python формально складається з усіх символів Unicode, хоча обмеження на використання символів, звісно, існують. Навіть деякі ASCII-символи (\$, ?, `) мають доволі обмежене використання¹⁰.

Серед символів алфавіту традиційно виділяють певні групи. У мовах програмування найчастіше виділяють **десяткові цифри**, **літери (букви)** та **порожні**, або **білі**, символи.

Як зазвичай, арабськими десятковими цифрами є 0, 1, 2, ..., 9. Літерами у мові Python є не тільки великі й малі латинські A, B, ... Z, a, b, ... z та символ підкреслення `_`, але й букви національних алфавітів та деякі інші категорії символів Unicode (див. PEP 3131).

Так звані **порожні**, або **білі**, **символи** не мають зображення й з'являються в тексті програми, якщо, створюючи його, натискати, наприклад, клавіші **Space** (пробіл), **Enter** («кінець рядка») і **Tab** (символ табуляції)¹¹. Порожні символи переважно використовуються в якості роздільників лексем (але не тільки) і, у мові Python за трьома винятками, самостійно лексеми не формують.

Серед лексем мови Python виділяють такі категорії лексем: **ідентифікатори** (*identifier*), **ключові слова** (*keyword*), **оператори** (*operator*), **роздільники** (*delimiter*) та **літерали** (*literal*). Крім зазначених класів лексем є ще три спеціальні лексеми: **NEWLINE**, **INDENT** та **DEDENT**.

Розглянемо зазначені категорії лексем більш детально.

4.2 Ідентифікатори та ключові слова

До 2007 року ідентифікатори могли містити тільки символи ASCII-діапазону.

⁹ Більш точно: у цій ситуації генерується (у документації до Python цьому відповідає англійське *raise*) виняток типу **SyntaxError**, але про винятки мова піде пізніше.

¹⁰ Ці символи можна використовувати тільки всередині коментарів та рядкових констант.

¹¹ Тут наведено не вичерпний перелік білих символів.

- **Ідентифікатор** (або **ім'я**) – непорожня послідовність символів, що складається з латинських літер (великих та малих), символу підкреслення та десяткових цифр, але починається не з цифри.

Довжина ідентифікаторів не обмежується. Великі й малі букви в іменах **відрізняються**, тобто регістр букв має значення. Так, **Python**, **PYTHON**, **python** є трьома різними ідентифікаторами. Також ідентифікаторами є: **abra8cadabra**, **mixedCase**, **CamelCase**, **snake_notation**, **x1**, **y2_2**.

Зараз Python 3 дозволяє використовувати в ідентифікаторах не тільки латинські букви, але й букви інших національних алфавітів (інші букви Unicode, зокрема букви, що не належать US-ASCII). Зокрема допускаються букви національних алфавітів та деякі інші символи, хоча використовувати ці символи в ідентифікаторах не рекомендується. Правила використання цих символів регламентує PEP 3131.

Серед ідентифікаторів виділяють **ключові слова**: зарезервовані слова, які мають спеціальний зміст (визначається мовою програмування). Використовувати ключові слова за іншим призначенням заборонено. У Python ключовими словами є такі.

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

Починаючи з Python 3.10 додатково виділяють так звані «м'які ключові слова» (англ. *soft keywords*), які мають спеціальний визначений мовою програмування зміст тільки в певних контекстах. У Python 3.14 це **match**, **case**, **_**, **type**.

Деякі сукупності ідентифікаторів мають спеціальне призначення, хоча при цьому не є зарезервованими словами:

— імена, що починаються та закінчуються двома символами підкреслення, наприклад, **__name__**;

Ці імена визначені інтерпретатором та його реалізацією. Мають використовуватися виключно у відповідності до документації мови.

— імена, що починаються з символу підкреслення, наприклад, **_var**;

Такими іменами розробники зазвичай використовують, щоб повідомити, що це ім'я відноситься до «внутрішньої кухні» коду і не має використовуватися ззовні. Зокрема, такі імена спеціальним чином обробляються під час імпорту бібліотек, в яких означені.

— імена, що починаються з двох символів підкреслення, наприклад, **__field**.

Далі дізнаємось, що такі імена також обробляються спеціальним чином.

Починаючи з Python 3.10 з'явилися так звані «м'які ключові слова» (*soft keywords*), які в деяких контекстах ведуть себе як ключові слова, тобто мають визначене мовою програмування призначення, проте в інших допустимо їх використання як звичайних ідентифікаторів.

4.3 Оператори та роздільники

Як зазначалося раніше, оператори використовуються для позначення операцій, що мають виконуватися над значеннями. У документації до операторів та роздільників відносять такі лексеми:

+	-	*	**	/	//	%	@	<<	>>	
&		^	~	<	>	<=	>=	==	!=	:=
+=	-=	*=	**=	/=	//=	%=	@=	<<=	>>=	=
(	)	[	]	{	}	,	:	;	.	

Деякі з них вже зустрічалися нам раніше, деякі будуть розглядатися далі. Зазначимо, що операції можуть позначатися не тільки окремими символами, наприклад + або –, але й послідовностями інших символів, наприклад <=, або навіть ключовими словами, наприклад **and**.

Домовимось надалі операторами вважати позначення операцій, включаючи до переліку не тільки лексеми-оператори, але й ключові слова, що позначають операції.

Особливість лексичних аналізаторів полягає в тім, що під час розбиття входу на лексеми лексичний аналізатор намагається виділити якомога довшу лексему. Так, за фрагментом **for12+qw** буде виділено три лексеми: **for12** (ідентифікатор), **+** (оператор) і **qw** (ідентифікатор), незважаючи на те, що **for** в іншому контексті також може розглядатися як лексема.

Лексеми у вхідному тексті можуть розділятися не тільки операторами, але й роздільниками та «пропусками» (англ. *white space*), що є непорожніми послідовностями білих символів. Наприклад, у фрагменті **qw=we** лексеми **qw** та **we** розділено знаком **=**. Якщо дві лексеми, записані одна після іншої, утворюють іншу лексему, то вони мають розділятися білими символами. Наприклад, **fun and sun**. При цьому білі символи, розташовані між двома лексемами одного рядка вхідного тексту, самотійно лексеми не утворюють. Отже, у наведеному прикладі лексичний аналізатор «побачить» три лексеми: **fun**, **and**, **sun**.

4.4 Літерали

У тексті програми *літерали* позначають значення вбудованих типів.

Іноколи літерали за інерцією називають *константами*, маючи на увазі те, що вони позначають константні значення певних типів. Це не дуже точно: адже деякі константи позначаються ключовими словами (наприклад, ключові слова **False** та **True** позначають значення логічного типу **bool**), а деякі константні значення в тексті програми позначаються виразами.

У мові Python наявні літерали числових типів **int**, **float**, **complex** та типів **str** та **bytes**.

4.4.1 Літерали числових типів

Правила запису констант числових типів обговорювалися раніше під час знайомства з цими типами. Тому зараз наведемо тільки особливості, що стосуються літералів цих типів.

Перш за все зазначимо, що літерали числових типів **int**, **float**, **complex** позначають тільки певну частину значень відповідних типів, бо задають невід’ємні цілі значення, невід’ємні дійсні значення та уявну частину (також без знаку) комплексних значень. Решта константних значень відповідних типів може бути записана тільки як **вираз**.

Приклади

1. Запис **-13** з точки зору мови Python є виразом, що складається з унарного оператора – та цілого літерала **13**.

2. Запис **-12.3e+02** також є виразом, що складається з оператора – та дійсного літерала **12.3e+02**. ◀

Уявні літерали, або літерали типу **complex**, позначають виключно комплексні числа з нулевою дійсною частиною. Вони починаються з дійсного літерала або непорожньої послідовності десяткових цифр, за якими йде літера **j** або **J**. Наприклад, **5j**, **1.34J**, **10e-04j**. На відміну від математики, де для позначення уявної частини використовується літера **i**, у мові Python використовуються літери **j** та **J**. Решта комплексних констант (з ненульовою дійсною частиною та/або з від’ємною уявною частиною) може бути записана як вирази, наприклад: **1+1j**, **-7j**, **-3.14+2j**.

Для зручності запису багаторозрядних чисел у мові Python є можливість використовувати **роздільник груп розрядів** — символ підкреслення **_**. Роздільник може бути доданий між довільними двома цифрами числового літерала. Так, літерали **1_000_000**, **1_0_0_0_0_0_0**, **1000000** позначають одне те саме ціле значення. Неважко бачити, що перший сприймається людиною краще. Також роздільники дуже корисні при запису двійкових літералів, щоб

випадково не помилитися з кількістю розрядів, наприклад запис **0b1111_1111** сприймається краще за **0b11111111**. У літералах дійсного та комплексного типу роздільник груп розрядів також може зустрічатися.

4.4.2 Рядкові літерали

З погляду математики рядок (*string*) — це послідовність символів. У мові Python для збереження послідовностей символів призначений вбудований тип **str**; символи розглядаються як символи Unicode. Значення цього типу прийнято називати *рядками*, або *рядковими значеннями*. Рядкові літерали, що задають значення типу **str**, можуть записуватись різними способами.

Рядковий літерал можна записати в однократних подвійних лапках. Наприклад, літерал **"You are welcome!"** задає рядок **You are welcome!**. Лапки навколо рядка до рядкового значення не входять. *Довжиною* рядка є довжина послідовності символів, що утворює рядок. При цьому рахуються саме символи рядкового значення, а не кількість символів його запису в тексті програми і не кількість байтів, що займає відповідна послідовність символів Unicode. Так, лапки навколо рядка в його довжині не враховуються.

Запис у подвійних лапках — не єдиний спосіб запису рядків у тексті програми. Рядкові літерали можна записувати не тільки в подвійних лапках, але й в одинарних, наприклад, **'You are welcome!'**. При виборі типу лапок варто намагатися за можливістю використовувати в коді лапки одного стилю: або всюди одинарні, або всюди подвійні. При цьому відкриваюча та закриваюча лапки рядкового літералу мають бути однаковими.

Хоча в рядкове значення рядкового літерала можуть входити довільні символи з Unicode, але деякі з них можуть бути записані тільки в спеціальний спосіб: через послідовність символів, що починається із зворотної скісної. Наприклад, у тексті рядкових літералів, що вже були розглянуті, символ **** безпосередньо записати не можна; він має записуватись як послідовність ****.

Комбінації символів вхідного тексту програми, що починаються із **** та позначають один символ, прийнято називати *керуючими послідовностями*, або, неформально, *escape-послідовностями*.

Послідовності **'** та **"** у тексті рядкових літералів позначають символи **'** та **"**. Символ, перед яким у рядковому літералі стоїть зворотна скісна, інколи називають *екранованим*. Якщо рядковий літерал обмежено подвійними лапками, то символ **"** у середині літерала може записуватись тільки через escape-послідовність. Якщо літерал обмежено одинарними лапками, то аналогічне стосується символу **'**. Лапки іншого вигляду (що відрізняються від лапок на початку та в кінці рядкового літералу) можна записувати в літералі без екранування. Якщо в рядку треба записати один із символів **'** та **"**, то щоб його не екранувати, навколо рядка ставлять лапки іншого типу.

Керуючими послідовностями також задають й інші символи (далі наведено тільки деякі з них):

\n — символ, що задає перехід на новий рядок;

\t — табуляція.

Наприклад, **"Буде виведено на одному рядку.\nА це вже на наступному."**, **"12\n34\t5"**. Довжина останнього рядка дорівнює 7.

До речі, перехід на новий рядок в рядковому літералі в однократних лапках, а саме такі розглядалися вище, можна задати тільки керуючою послідовністю.

Керуючі послідовності використовуються і для того, щоб задати символ із певним кодом. Деякі керуючі послідовності та їх значення наведені в табл. 8.

Табл. 8. Керуючі послідовності та їх значення

Керуюча послідовність	Що позначає	Приклади, примітки	
<code>\ooo</code>	Символ з вісімковим кодом <code>ooo</code> .	<code>\040</code> пробіл (його код 32) <code>\061</code> цифра 1 <code>\70</code> цифра 8 <code>\101</code> латинська A <code>\141</code> латинська a Може зазначатися не більше трьох розрядів.	
<code>\xhh</code>	Символ з шістнадцятковим кодом <code>hh</code> .	<code>\x31</code> цифра 1 Має зазначатися рівно 2 розряди.	
<code>\uxxxx</code>	Символ з шістнадцятковим кодом <code>xxxx</code> .	Має зазначатися рівно 4 розряди.	
<code>\Uxxxxxxxx</code>	Символ з шістнадцятковим кодом <code>xxxxxxxx</code> .	Має зазначатися рівно 8 розрядів.	

Нагадаємо ще раз: довжина рядка визначається не за кількістю символів запису відповідного рядкового літералу, а за кількістю символів відповідного рядкового значення. Так, довжина рядка `"\070"` дорівнює 1, бо рядковим значенням є послідовність, що складається з єдиного символу 8. Рядок, що не містить жодного символу, називають порожнім. Довжина порожнього рядка дорівнює 0. Задати його можна, наприклад, так `""`.

Рядкові літерали можна записувати в трикратних лапках (*triple-quoted string*), наприклад:
`""""Ця рядкова константа`
записана на двох рядках і буде виведена також на двох рядках.`""""`

В якості лапок можна використовувати, як символ `"`, так і символ `'`, але лапки, що розпочинають рядкову константу та що її завершують, мають бути однаковими.

Відмінністю від рядкових літералів в однократних лапках є те, що символ кінця рядка всередині рядкового значення обробляється буквально і входить до відповідного рядкового значення. Так, літерали

`""""Перша частина.`
`-Друга частина. """"`

та
`"Перша частина.\n-Друга частина."`
задають одне те саме рядкове значення.

Лапки, що обмежують рядок, можуть записуватися без екранування, якщо їх не більше двох посліпль. Наприклад,

`"""" Приклад рядка з лапками " та ""`. І це є коректно записана рядкова константа.`""""`

Літерали в трикратних лапках найчастіше одноосібно займають рядок або кілька рядків тексту. Для них в якості обмежувачів рекомендують використовувати три подвійні лапки.

У деяких ситуаціях кількість символів рядка, які треба екранувати, може бути такою, що відповідний літерал буде погано сприйматися людиною. Виходом тут є використання так званих *сирих (необроблених) рядків* (англ. *raw string*), в яких усі керуючі послідовності сприймаються буквально. На початку сирого рядка має вказуватися префікс `r` або `R`. Лапкам, що збігаються з лапками-обмежувачами, має передувати зворотна скісна. Так, запис `r"12"3"` є синтаксично правильним та позначає рядок `12"3`; запис `r"12"3"` містить синтаксичну помилку: лексичний

аналізатор буде розглядати його як три лексеми **r"12"**, **3** та **"**. Ще одне обмеження сирих рядків: вони не можуть завершуватися однією зворотною скісною (бо лексичний аналізатор тоді не буде сприймати заключні лапки як ознаку завершення рядкової константи): запис **r"\"** помилковий, проте **r"\"** задає рядкове значення `\\`.

Крім зазначених різновидів літералів, що позначають у програмі рядкові значення, мова Python ще використовує **відформатовані рядки**, або **f-рядки** (англ. *formatted string literal*), рядкове значення яких утворюється в результаті форматування даних. Ознакою відформатованого рядка є префікс **f** або **F** на початку. Ці префікси можуть комбінуватися з префіксами **r**, **R**; порядок та регістр префіксів кожної із груп неважливий: кожна із комбінацій **fr**, **Fr**, **fR**, **FR**, **rf**, **rF**, **Rf**, **RF** є допустимою і позначає, що далі записано сирий відформатований рядок.

Деталі відформатованих рядків та операції, застосовні до рядків типу **str**, будуть розглянуті пізніше.

Кілька записаних поспіль (можливо, через білі символи) на одному рядку рядкових літералів утворюють одне значення. При цьому рядкове значення другого літералу конкатенується (дописується в кінець) з рядковим значенням першого літералу. Наприклад, за послідовністю літералів **"Hello"** **"World"** буде утворене те саме значення, що й за літералом **"HelloWorld"**.

Цей прийом може бути корисним, коли треба записати довгий літерал: розбивши його на частини можна уникнути занадто довгих рядків тексту, що не вміщуються в ширину монітору; крім того, цей прийом дозволяє записати літерал на кількох рядках та написати окремий коментар поруч з кожною його частиною. Також таке розбиття на частини має сенс, коли різні частини рядка необхідно записати рядковими літералами різного вигляду, наприклад, у вигляді сирого, відформатованого та звичайного рядка, через неможливість зручно записати одним рядковим літералом.

Крім рядкових літералів мова Python має літерали для зображення послідовностей байтів, що зберігаються як значення типу **bytes**. Такі константи мають розпочинатися з префіксів **b** або **B**. Ці послідовності ззовні, тобто поведінкою, доволі схожі на рядки. Навіть у багатьох книжках вони розглядаються одночасно з типом **str**. Але якщо дивитися глибше, то відмінності більш ніж суттєві, наприклад, через те, що тип **bytes** призначений для маніпулювання двійковою інформацією. Він може зберігати послідовності символів, але дивиться на окремі символи як на байти, тобто його можливості обмежені послідовностями символів з таблиці ASCII; натомість рядки типу **str** зберігають послідовності символів Unicode. Більш детальне знайомство з літералами, що задають значення типу **bytes**, і з самим цим типом відкладемо до того моменту, коли використання типу **bytes** дійсно стане доцільним у наших програмах.

Тип символів у мові Python відсутній. За потреби символи можна зображувати рядками довжини **1**.

4.5 Коментарі¹²

Частина тексту програми, що починається символом **#** (*hash*), що не є частиною літерала, і завершується перед першою послідовністю символів, що задає ознаку кінця рядка текстового файлу, інакше безпосередньо перед кінцем файлу, вважається **коментарем**¹³. Під час виконання коментарі ігноруються, лексеми за ними не створюються. При цьому коментарі є корисними для

¹² Починаючи з Python 3.12 це не єдиний випадок, коли символ **#** розпочинає коментар, але зараз ці випадки розглядати не будемо.

¹³ У деяких неофіційних джерелах стверджують, що коментар замінюється пробілом; але на семантику мови це уточнення не впливає.

програміста. Вони зазвичай використовуються програмістами для пояснення призначення коду або певні його особливості, щоб полегшити роботу з цим кодом у майбутньому.

Наприклад.

```
pi = 3.14 # Для поточних обчислень такої точності достатньо.
```

У наведеному прикладі коментар підкреслено.

Коментар, тотожний змісту коду та/або такий, який просто «перекладає» наявні в коді інструкції на природню мову, тобто постулює очевидний факт, є *надлишковим*, наприклад.

```
radius = 2.0 # Ім'я radius позначає радіус.
```

Це типовий приклад надлишкового коментаря. *Надлишкових коментарів слід уникати*. Вони зовсім не прояснюють призначення коду, а, навпаки, ускладнюють читання коду програмістом.

PEP8 дає такі рекомендації щодо коментарів.

- Коментарі мають складатися з повних речень, кожне з яких починається з великої літери (якщо не є ідентифікатором). Якщо коментар складається з кількох речень, то кожне з них має завершуватися крапкою і між ними має бути не один, а два пробіли.

- Коментарі слід писати англійською мовою. Національну мову можна використовувати тільки тоді, коли є впевненість на 120%, що відповідний код будуть читати тільки ті, хто цю національну мову розуміє.

- Якщо рядок містить не тільки коментар, то перед символом #, що розпочинає коментар, має бути принаймні два пробіли.

Багаторядкові коментарі у мові Python відсутні. За потреби записати такий коментар використовують рядки в трикратних лапках. Але формально коментарями вони не є.

4.6 Логічні рядки та спеціальні лексеми

Текстові файли прийнято поділяти на рядки. Ознакою кінця рядка є спеціальна послідовність символів, що записується в кінці рядка. Різні операційні системи можуть використовувати різні ознаки кінця рядка, наприклад, **LF** (символ з кодом 10, Line Feed), або **CR** (символ з кодом 13, Carriage Return), або комбінацію **CR LF**. Також ознакою кінця рядка вважається фізичний кінець входу.

Файл з текстом програми також є текстовим. Інтерпретатор розглядає його рядки як *фізичні рядки*. Зазначимо, що лексичний аналізатор Python вміє правильно обробляти всі наведені ознаки кінця рядка. Під час оброблення фізичні рядки об'єднуються в *логічні*. Кожен логічний рядок складається з одного чи кількох фізичних рядків; у кінець кожного логічного рядка лексичний аналізатор додає спеціальну лексему **NEWLINE**.

- Лексема **NEWLINE** утворюється тільки як ознака кінця логічного рядка.

Фізичні рядки можуть об'єднуватися в логічні *явно* або *неявно*. Якщо фізичний рядок завершується неекранованою зворотною скісною (\), що не є частиною коментаря, і після якої йде тільки ознака кінця рядка, то під час лексичного аналізу ця зворотна скісна разом з послідовністю, що задає фізичний кінець рядка, ігнорується. Тим самим відбувається *явне* з'єднання двох фізичних рядків, які разом входять в один логічний рядок. У такий спосіб можуть з'єднуватися не тільки два, але й більше фізичних рядків. Наприклад, послідовність фізичних рядків

```
radius =\
15.2
```

буде об'єднана в рядок.

```
radius = 15.2
```

Послідовність фізичних рядків

```
"123\
45"
```

буде оброблятися як єдиний рядок.
"12345"

Водночас є певні обмеження: за допомогою зворотної скісної не можна розривати лексеми (за винятком літералів, що задають значення типів **str** або **bytes**). Також у такий спосіб не можна продовжити коментар.

Неявне з'єднання фізичних рядків відбувається, коли:

— рядковий літерал в трикратних лапках (англ. *triple-quoted string*) розташовано на кількох рядках; наявні всередині символи фізичного кінця рядка на відміну від явного об'єднання при цьому зберігаються; наприклад, послідовність фізичних рядків

```
"""  
first line of string  
second line of string  
"""
```

буде об'єднана в логічний рядок

```
"""\nfirst line of string\nsecond line of string\n"""
```

— вираз у дужках (круглих, квадратних, фігурних) записаний на кількох рядках тексту, наприклад послідовність фізичних рядків

```
[1, 2, 3,  
    4, 5]
```

буде оброблятися як один логічний рядок.

Зауважимо, що в наступному прикладі є **помилка**: неявне з'єднання рядків відбувається тільки всередині виразів, записаних у дужках.

```
smth =          # перший логічний рядок, містить синтаксичну помилку  
    [1, 2, 3,    # частина другого логічного рядка  
      4, 5]      # частина другого логічного рядка
```

У той же час послідовність фізичних рядків

```
smth = [  
    1, 2, 3,  
    4, 5  
]
```

утворює один логічний рядок, є синтаксично коректною і навіть відповідає стильовим рекомендаціям PEP8.

Робота інтерпретатора з кодом мовою Python починається з того, що він послідовно обробляє логічні рядки.

Кожна інструкція має записуватися на одному логічному рядку за винятком випадків, де лексема **NEWLINE** дозволяється синтаксисом (наприклад, у складеній інструкції). Ситуація, коли на одному рядку записується кілька інструкцій, можлива, але мові Python не притаманна.

Оброблення білих символів. Білі символи, розташовані поза лексемами логічного рядка, ігноруються. Логічний рядок, що не містить нічого, крім білих символів та коментаря, під час оброблення ігнорується.

Білі символи (пробіли та табуляції), що розташовуються на початку логічного рядка, можуть породжувати лексеми **INDENT** та **DEDENT**, які задають рівень абзацного відступу рядка (англ. *indentation level of the line*) і тим самим визначають групування інструкцій коду.

За збільшення кількості початкових білих символів логічного рядка (відносно попереднього логічного рядка) рівень на 1 збільшується і генерується лексема **INDENT**, за зменшення (і повернення на попередній рівень) рівень на 1 зменшується і генерується лексема **DEDENT**.

Під час оброблення білих символів на початку логічного рядка табуляції замінюються пробілами (від 1 до 8, щоб загальна кількість символів, включаючи підставлені, стала кратною 8). Якщо табуляції вживаються одночасно з пробілами так, що зміст залежить від кількості пробілів, що відповідають табуляціям, то це є помилкою.

PEP8 рекомендує для кожного наступного рівня відступу додавати чотири пробіли.

Для першого рядка файлу вважається, що відступ попереднього був нулевим. Також додамо, що абзацний відступ не може розбиватися на кілька фізичних рядків.

Лексеми **INDENT** та **DEDENT** задають рівень вкладеності інструкцій. Правила їх використання регламентуються синтаксисом інструкцій мови. Інструкції мови, що мають однаковий рівень вкладеності, в однопотоковій програмі виконуються поспіль і мають записуватися з однаковим абзацним відступом. Ситуація, вигляду

інструкція1

інструкція2

інструкція3

інструкція4

розглядається інтерпретатором приблизно так: під час виконання інструкції 1 виконуються інструкції 2 та 3, далі виконання інструкції 1 завершується і починає виконуватися інструкція 4.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: лексема, токен, ідентифікатор, ключове слово, оператор, роздільник, літерал, константа, рядок, рядкове значення, довжина рядка, керуюча послідовність, escape-послідовність, екранування символів, сирий рядок, коментар, надлишковий коментар, логічний рядок, фізичний рядок, **NEWLINE**, **INDENT**, **DEDENT**

2. У чому полягає задача лексичного аналізатора?
3. У якому кодуванні за умовчанням записується текст програм мовою Python?
4. З яких символів будуються ідентифікатори?
5. Чи може програміст записувати власні ідентифікатори в тексті програми?
6. Чи може програміст додати нове ключове слово?
7. Яке призначення лексем-операторів?
8. Яке призначення лексем-роздільників?
9. Чи можуть оператори та роздільники записуватися кількома символами?
10. Як лексичний аналізатор обробляє ситуацію, коли одна лексема може розглядатися як початок іншої?
11. У чому відмінність літералів від констант?
12. Літерали яких типів наявні в мові Python?
13. Чи можна записати літерал числового типу, що позначає від'ємне значення?
14. У яких системах числення можна записувати літерали цілого типу?
15. Як має записуватися дійсний літерал?
16. Як має записуватися літерал, що задає комплексне значення?
17. Назвіть ситуації, коли доречно використання роздільника груп розрядів?
18. У літералах яких числових типів може використовуватися роздільник груп розрядів?
19. Як записуються рядкові літерали?
20. Чи можна записувати лапки всередині рядкового літералу? Якщо так, то за яких умов і як?
21. Як записати літерал для рядка, що містить символ переходу на новий рядок?
22. Як записати літерал для рядка, що містить зворотну скісну?
23. Через що лексичний аналізатор не завжди може правильно обробляти неекрановані символи ' , " , \, записані всередині рядкових літералів?
24. В яких випадках доцільно використовувати сирі рядки?
25. Рядкові літерали в однократних та трикратних лапках позначають значення одного того самого типу чи різних?

26. Як співвідносяться між собою довжина рядка та довжина літералу, що задає відповідний рядок? Відповідь поясніть.

27. Чи можна в рядковому літералі задати символ безпосередньо за його кодом?

28. Чи можна один рядок записати кількома літералами?

29. Чи наявний серед вбудованих типів мови Python тип для зображення символів?

30. Як можна записати в програмі окремих символ?

31. Для чого в програмах використовуються коментарі?

32. Як коментар оформлюється синтаксично?

33. Чи може один коментар бути записаний на кількох фізичних рядках тексту програми?

34. Яка лексема позначає кінець рядка?

35. В яких ситуаціях кілька фізичних рядків можуть неявно об'єднуватися в один логічний?

36. Як явно задати об'єднання фізичних рядків в один логічний?

37. Як обробляються рядки, що складаються виключно з білих символів та коментарів?

38. Як обробляються білі символи на початку логічного рядка?

39. Як обробляються білі символи в кінці логічного рядка?

40. Які лексеми генеруються за зміною абзацного відступу?

41. За якими правилами генеруються лексеми **INDENT** та **DEDENT**?

42. Чи можна в тексті програми на початку фізичних рядків навмання записувати білі символи?

43. На яку послідовність символів згідно рекомендацій PEP8 має збільшуватися абзацний відступ на початку рядка тексту програми, якщо треба задати збільшення рівня вкладеності?

Задачі та вправи

1. Які лексеми "побачить" лексичний аналізатор у рядку тексту (якщо поділить його на лексеми до кінця, а не буде зупинений синтаксичним аналізатором на першій синтаксичній помилці)?

а) `x2=3 for x 2 = 3 y<= 4.67E+02 "for2 #" for2 # smth`

б) `2 >= x2 'while' 3 -4.67 "for2" for2 #smth # x = y`

2. Що з наведеного нижче переліку не є однією лексемою (є кількома лексемами або містить лексичну помилку)?

`<=, +=1, 1+1j, 2for, a_or_b, "c:\\install", "q'q", "q"q", ""q""q""", "" q" q""", r"\`

3. Записати літерали, значеннями яких є рядки довжиною 1, що складаються з символів ' , " , \ .

4. Записати літерал, значенням якого є рядок '-"-.\.

5. Чи є різниця між значеннями "a" та 'a'?

6. Поясніть різницю між а) **123** та **"123"**; б) **12** та **12.** .

7. Записати літерал, значенням якого є `c:\install\python`. Виконати завдання з та без використання сирих рядків.

8. Записати літерал, значенням якого є

`\*`
приклад одиничних лапок ' ''
приклад подвійних лапок " ""
`*\`

9. Записати літерал, значенням якого є

Програма поки тільки повідомляє свого автора.

Виконавець: прізвище

ім'я

по батькові

Група: К-хх

Замість *ПБ* та *xx* підставити власну інформацію. Кількість пропусків має бути відповідна до завдання.

5 Дані в пам'яті програми

Одиницею збереження даних у пам'яті програми мовою Python є об'єкт. Доступ програміста до оброблюваних даних забезпечується через змінні програми. У цьому розділі переважно розглядатиметься організація збереження оброблюваних даних у пам'яті під час виконання програми та поняття змінної. Найчастіше (але не завжди) змінні утворюються в результаті виконання інструкції присвоювання. Насправді присвоювання має дещо ширші можливості, які також будуть розглянуті.

5.1 Об'єкти, або Зображення даних у пам'яті

Під час виконання значення, тобто дані, мають якимось зберігатися в пам'яті. У мові Python абстракцією даних є об'єкт: дані зберігаються у вигляді об'єктів, що розміщуються в пам'яті програми та мають тип, значення та власний ідентифікатор¹⁴ (англ. *identity*). Ідентифікатор є цілим числом; у CPython під ним можна розуміти адресу, за якою об'єкт розташовано в пам'яті. Ідентифікатор об'єкта встановлюється під час його створення і далі не змінюється. Об'єкти, що одночасно розміщено в пам'яті, завжди мають різні ідентифікатори. Тим не менш, якщо час існування об'єктів не перетинається, то їхні ідентифікатори можуть співпадати. Об'єкти, що одночасно існують у певний момент часу, вважаються **різними**, якщо мають різні ідентифікатори; при цьому різні об'єкти можуть мати однакове значення. Тип об'єкта визначає допустимі операції над ним та також за звичайних обставин встановлюється під час створення об'єкта і далі не змінюється.

Значення об'єкта. Перш за все, не слід ототожнювати вміст ділянки пам'яті, де розташовано об'єкт, зі значенням об'єкта. Слід згадати тезу з документації: «Об'єкт – абстракція даних». Це означає, що об'єкт варто розуміти як ключ до даних, що надає до них доступ та дозволяє їх обробляти. При цьому де та як (в якому форматі) зберігаються дані, які зображує об'єкт у програмі, визначається реалізацією інтерпретатора. Зокрема, ніхто не стверджує, що всі дані, за які відповідає об'єкт, зберігаються безпосередньо в цьому об'єкті. Крім того ділянка пам'яті, в якій розташовано об'єкт, може містити службову інформацію, що визначається реалізацією інтерпретатора.

Для об'єктів вбудованих типів **int**, **float**, **complex**, **bool**, **str**, **bytes** (а також деяких інших) природньо ототожнювати значення об'єкта та значення зображуваних цим об'єктом даних. Але така оптимістична картина відбувається не для всіх типів. Доволі складний випадок виникає, коли значення, що зберігається, структурно утворюється з інших значень (наприклад, як послідовність¹⁵ кількох значень). У такому разі (див. рис. 2) можна уявляти, що такий об'єкт зберігає тільки посилання на об'єкти, що зберігають частини зображуваних ним даних.

Також можна уявляти, що ці посилання зберігаються як ідентифікатори відповідних об'єктів (у випадку CPython можна казати про адреси). Об'єкти, що містять посилання на інші об'єкти, у мові Python називають **контейнерами**.

¹⁴ Дослівний переклад – «ідентичність».

¹⁵ Зараз не будемо акцентуватися на назвах конкретних типів мови Python.

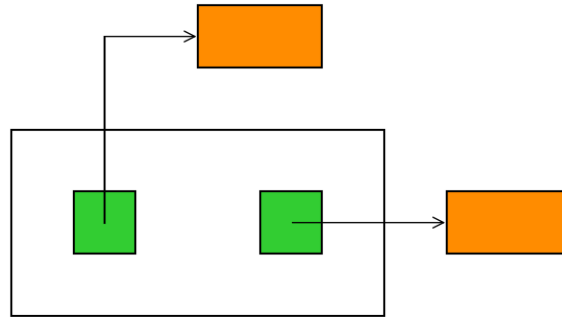


Рис. 2 — Схематичне зображення об'єкта-контейнера (зеленим виділено посилання на об'єкти, що зберігають частини значення)

Виникає природне питання: що вважати значенням об'єкта, якщо він є контейнером? З одного боку, під значенням природньо розуміти **значення зображуваних ним даних**, тобто значення, що складається зі значень його частин-об'єктів, — і в більшості випадків саме таке розуміння використовується. Але з іншого боку, тут також можна розглядати значення, що містить посилання на його частини, а не самі частини безпосередньо; будемо розуміти його як **значення об'єкта**.

Хоча така термінологія в офіційній документації явно не використовується, але дозволяє уникнути «перлів» про «змінювані незмінювані значення».

Якщо після створення об'єкт може змінювати власне значення, то його вважають **змінюваним** (англ. *mutable*), інакше — **незмінюваним** (англ. *immutable*). Ще раз підкреслимо, що для об'єктів-контейнерів тут йдеться про значення, що складається з посилань на його частини. Також слід розуміти, що в мові Python **незмінюваний** та **сталий** (англ. *unchangeable*) не є синонімами. «Сталий» використовується в тому випадку, коли значення зображуваних об'єктом даних не може бути змінено.

Можлива ситуація (див. рис. 3), коли дані під час виконання програми зображуються одним тим самим незмінюваним об'єктом і при цьому можуть змінюватись, тобто не є сталими.

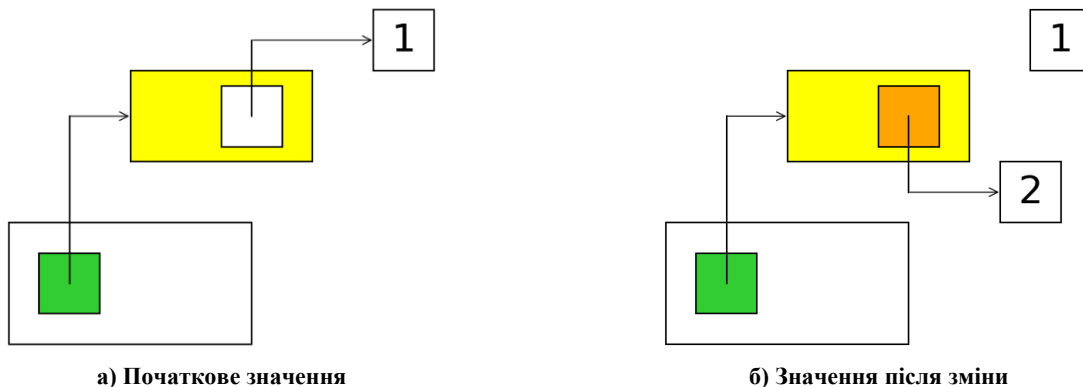


Рис. 3 — Схематичне зображення зміни значення незмінюваного об'єкта-контейнера

Наведена колізія можлива тільки для контейнерів.

Об'єкти числових типів **int**, **float**, **complex**, **bool** та рядкових типів **str**, **bytes** контейнерами не є, а також є незмінюваними: після створення відповідного об'єкта не може змінюватися ані значення об'єкта, ані значення відповідного типу даних, яке цей об'єкт зображує.

У стандартних випадках чи може змінюватися значення об'єкта визначається його типом, точніше означеними для нього операціями. Тому терміни змінюваний/незмінюваний інколи переносять з об'єктів на типи. Під змінюваним типом мають на увазі тип, значення якого зображуються змінюваними об'єктами. Аналогічно розглядають незмінювані типи. Отже, типи

int, float, complex, bool, str, bytes є незмінюваними. Знайомство зі змінюваними об'єктами та відповідними типами відбудеться пізніше.

У мові Python у вигляді об'єктів зображуються не тільки значення певних типів, але й самі типи та не тільки. Не буде помилкою сказати, що в пам'яті все зображується об'єктами.

5.2 Поняття змінної в інформатиці

Повернемося більш детально до поняття змінної. Як зазначалося раніше, під змінною величиною можна розуміти загальне поняття реального чи уявного об'єкта (або його окремої характеристики), який може перебувати в різних станах.

У книжках з програмування та з інформатики, у документації на мови програмування можна прочитати багато різноманітних означень того, що таке змінна, які уточнюють, доповнюють одне одного та/або відображають особливості певної мови програмування. У різних парадигмах програмування означення змінної можуть різнитися, навіть доволі суттєво.

В імперативному програмуванні можна побачити такі означення змінної (більшість з них для мови Python незастосовна):

- модель (зображення) об'єкта в пам'яті комп'ютера;
- абстракція однієї чи кількох комірок пам'яті комп'ютера;
- іменована ділянка пам'яті;
- ділянка пам'яті (у машинній програмі), яка може мати різні стани й зображувати ними різні стани об'єкта;
- елемент програми, що зберігає інформацію;
- іменована, або адресована інакше, ділянка пам'яті, адресу якої можна використовувати для здійснення доступу до даних і змінювати значення під час виконання програми.

Зауваження. Із розвитком мов програмування поняття змінної теж змінювалось. Настільки, що в документації деяких популярних мов програмування термін «змінна» знайти майже неможливо.

Більш коректним буде казати не про означення терміну «змінна» в програмуванні, а про властивості змінних. На властивості змінних можна дивитись з *точки зору тексту програми* та з *точки зору процесу виконання цієї програми*.

У тексті програми змінні позначаються (ідентифікуються) **іменами**, або **ідентифікаторами**. В імперативних мовах програмування під час виконання змінній відповідає певна сутність, що зберігає її **значення**.

Змінна може мати **тип**, що задає формат зберігання її значення, методи маніпуляції та трактовки вмісту. Мови, в яких кожна змінна має тип, вважають **типізованими**. Якщо в усіх випадках тип змінної може бути визначений на етапі трансляції (до виконання), то таку типізацію називають **статичною**. Якщо тип змінної визначається тільки під час виконання, то це **динамічна** типізація. **Безтиповими**, наприклад, є машинні мови.

Над змінними можуть виконуватися такі операції, як **встановлення значення змінної**, або **присвоювання**¹⁶, та **добування значення змінної**, або **розіменування** (імені змінної).

¹⁶ У контексті конкретної мови програмування присвоювання значення змінній може відбуватися не тільки в результаті виконання інструкції або операції присвоювання відповідної мови.

5.3 Змінні в мові Python

Тепер настав час поєднати між собою змінні та об'єкти мови Python. У тексті програми змінна асоціюється зі своїм іменем і позначає певну сутність, яка матеріалізується під час виконання програми.

У мові Python визначальними атрибутами змінної є ім'я та значення. Ім'я змінної — це ідентифікатор, що позначає змінну (певну сутність) в програмі. Значення змінної під час виконання моделюється об'єктом.

«Життя» змінної починається з того, що її ім'я зв'язується з деяким об'єктом. Іменем змінної має бути ідентифікатор, але не ключове слово. Найпростішим випадком операції зв'язування імені з об'єктом (англ. *name binding operation*) є виконання інструкції **присвоювання** (англ. *assignment*).

У найпростішому випадку синтаксис присвоювання такий:

ідентифікатор = вираз

Припустимо, що змінних з іменами **n** та **m** зараз не існує. Створимо змінну **n**. Для цього запишемо.

```
n = 13
```

За виконання цієї інструкції буде обчислено вираз праворуч від знаку **=**. Вираз складається з літерала, тому за ним буде утворено ціле значення та збережено в деякому об'єкті, далі цей об'єкт буде зв'язано з іменем **n**. Тепер ім'я **n** вважається означеним (англ. *defined*), змінну **n** можна вважати створеною і вона позначає ціле **13**. Схематичне зображення стану пам'яті після виконання присвоювання див. на рис. 4),



Рис. 4 — Схематичне зображення стану пам'яті

Означені імена можна розіменовувати, наприклад, використовувати у виразах. **Результатом розіменювання імені є об'єкт, з яким це ім'я зв'язане**; значення змінної отримується як значення, що зберігає відповідний об'єкт. За виконання

```
m = 2 * n
```

буде обчислено вираз праворуч, при цьому буде розіменовано змінну **n**, операція множення утворить нове значення — ціле **4**, що є результатом обчислення усього виразу праворуч від знаку **=**, це значення буде збережено в об'єкті і цей об'єкт буде зв'язано з іменем **m**. Утвориться змінна **m** (стан пам'яті див. рис. 5),

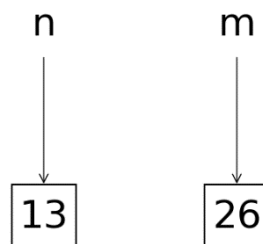


Рис. 5 — Схематичне зображення стану пам'яті

Упродовж роботи програми одне те саме ім'я може зв'язуватися з різними об'єктами. За виконання

`n = 15`

буде створено об'єкт, що зберігає ціле **15**, і цей об'єкт буде зв'язано з іменем **n** (стан пам'яті див. рис. 6). Значення змінної **n** змінилося! Зазначимо, що об'єкт з цілим **13** ще може продовжувати існування.

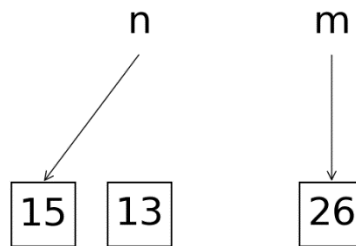


Рис. 6 — Схематичне зображення стану пам'яті

Отже, інструкція присвоювання дозволяє як утворити змінну та надати їй початкове значення (якщо змінної не існувало), так і змінити значення вже існуючої змінної (точніше, раніше означеного імені). Зміна значення змінної може відбуватися як у результаті зміни значення об'єкта (приклади будуть розглянуті під час розгляду змінюваних типів даних), так і в результаті повторного зв'язування (англ. *rebinding*) імені змінної (як у наведеному прикладі). Турбуватися про незмінюваність числових типів тут не слід: зміна значення змінної відбувається за рахунок зв'язування імені змінної з іншим об'єктом.

Продовжимо приклад. Після виконання

`n = 5.5`

ім'я **n** буде зв'язане з об'єктом, що зберігає дійсне **5.5** (див. рис. 7). Змінилося не тільки значення змінної, але й її тип. І в мові Python ці дії є синтаксично легальними.

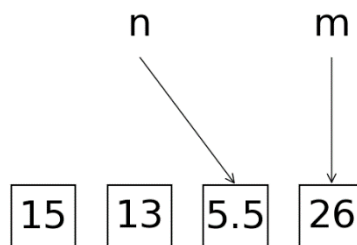


Рис. 7 — Схематичне зображення стану пам'яті

У результаті виконання присвоювання

`n = m`

буде обчислено вираз праворуч від знаку `=`: результатом обчислення буде об'єкт, зв'язаний зі змінною **m** (див. рис. 8). Далі цей об'єкт буде зв'язано з іменем **n**. Змінна **n** отримає значення **26**.

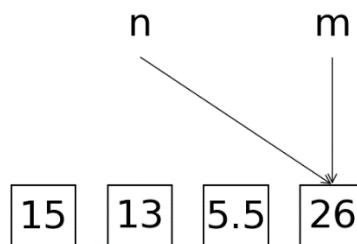


Рис. 8 — Схематичне зображення стану пам'яті

Продовжимо приклад. Нехай далі виконується
`m = 10`

За цієї інструкції змінюється не об'єкт, який зв'язано з іменем **m**, а ім'я **m** зв'язується з іншим об'єктом. Тому значенням змінної **n** залишиться **26**, а значення змінної **m** зміниться та стане **10** (див. рис. 9).

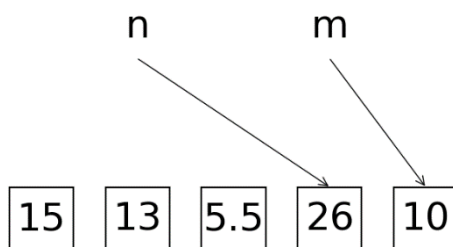


Рис. 9 — Схематичне зображення стану пам'яті

Наприкінці додамо, що розіменовувати неозначені імена (тобто ті, що не були зв'язані з об'єктами) не можна. Спроба це зробити може призвести до аварійного завершення програми.

Зараз можемо сказати, що таке змінна в Python з точки зору процесу виконання програми:

змінна в Python — це ім'я, під час виконання зв'язане з деяким об'єктом у пам'яті.

Зауваження. У програмуванні доволі часто кажуть про **ініціалізацію змінних** — встановлення початкового значення змінної (зазвичай під час її створення). Якщо в деяких мовах створювані змінні можуть залишатися без початкового значення, то в мові Python змінні завжди ініціалізуються, бо створення полягає в зв'язуванні імені змінної з її значенням.

- Мова Python є динамічно типізованою мовою. Тип змінної відповідає типу її значення. Під час існування змінна мови Python може змінювати не тільки значення, але й тип та місце розташування значення в пам'яті.

Невеликий експеримент. Раніше зазначалося, що в мові Python усе зображується об'єктами, у тому числі й типи даних, а не тільки звичайні дані (наприклад, числові). Проведемо експеримент в інтерпретаторі (див. табл. 9). Попередньо нагадаємо, що значенням виразу, що складається з ідентифікатора, є об'єкт, що його зв'язано з цим ідентифікатором.

Табл. 9 — Типи у мові Python

Вікно інтерпретатора	Примітки
>>> type(type(1))	Вихід інтерпретатора свідчить про те, що результат обчислення типу також зображується об'єктом класу type .
<class 'type'>	
>>> type(int)	І тип int у пам'яті зображується об'єктом класу type
<class 'type'>	
>>> type(type)	Більш того, клас type в пам'яті також зображується об'єктом класу type . 😊
<class 'type'>	

Час існування об'єктів. У мові Python доречно казати про час існування об'єктів. Об'єкти створюються тоді, коли породжується нове значення. Нове значення породжується за літералом, у результаті обчислення виразу з арифметичними операціями тощо. Якщо вираз складається тільки з імені змінної, то за його обчислення нове значення не породжується, а результатом обчислення є існуючий об'єкт, що зв'язаний з цим іменем.

Під час виконання може бути ситуація, що деякі об'єкти зв'язані з кількома змінними, а деякі з жодною. Деякі об'єкти можуть бути зв'язані з змінними не безпосередньо, а через інші об'єкти

(наприклад, об'єкти, що є частинами контейнера). Як би не було, але існуючі в певний момент часу об'єкти поділяються на ті, до яких у програміста є доступ (об'єкт може бути отриманий у результаті обчислення виразу, що не породжує нових значень), та на решту, які є недосяжними (англ. *unreachable*) для програміста. По суті недосяжні об'єкти є **сміттям** (англ. *garbage*): пам'ять займають, але використати їх неможливо.

У мові Python нема явних інструкцій для знищення об'єктів, натомість відбувається автоматичне **збирання сміття** (англ. *garbage collecting*): звільнення пам'яті з-під недосяжних об'єктів та повернення звільненої пам'яті в загальний обіг. Конкретний алгоритм збирання сміття залежить від реалізації інтерпретатора, більш того, документація дозволяє інтерпретатору відкладати збирання сміття навіть до кінця роботи програми, чи навіть узагалі його не виконувати, якщо пам'яті для розміщення нових об'єктів вистачає. Поточна реалізація CPython використовує метод, що обчислює кількість посилань на об'єкти та вміє враховувати циклічні посилання. Циклічні посилання (див. рис. 10) виникають тоді, коли невикористовувані об'єкти циклічно посилаються друг на друга і в результаті лічильник посилань кожного виявляється ненульовим, хоча доступу в програміста до цих об'єктів нема.

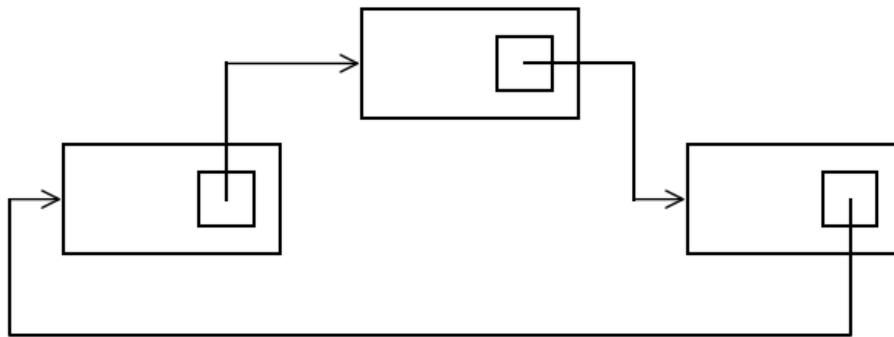


Рис. 10 — Схематичне зображення стану пам'яті

Інструкція `del`. За виконання інструкції `del ідентифікатор` відбувається розірвання зв'язку між іменем та зв'язаним з ним об'єктом, тобто знищення змінної; при цьому об'єкт, що був значенням змінної, може продовжувати існувати.

За виконання інструкції

```
n = 13.5
```

буде створено об'єкт, значенням якого є дійсне **13.5**, і цей об'єкт буде зв'язано з іменем **n**, тобто буде утворено змінну **n**.

За виконання інструкції

```
m = n
```

буде обчислено значення змінної **n** (це буде той об'єкт, що зберігає **13.5** і був утворений за виконання попередньої інструкції) і зв'язано з іменем **m**. Тепер один той самий об'єкт одночасно зв'язано з двома змінними.

Далі за виконання інструкції

```
del n
```

буде розірвано зв'язок між іменем **n** та позначуваним їм об'єктом. З цього моменту змінна **n** перестала існувати: тепер ім'я **n** нічого не позначає і розіменовувати його не можна. Проте її значення – об'єкт **13.5**, залишається в пам'яті та навіть є досяжним через ім'я **m**. Якщо далі виконати інструкцію

```
del m
```

то буде знищено змінну **m** (розірвано зв'язок імені **m** та позначуваного цим іменем об'єкта). Після цього об'єкт **13.5** став недоступним для програміста. Чи буде знищено сам об'єкт та

коли — відповідь на це питання знає тільки інтерпретатор. Може статися так, що цивілізоване знищення об'єкта збиральником сміття взагалі не відбудеться.

Зауваження. Розглянуту інструкцію можна ще записати так

del перелік імен змінних через кому

наприклад.

del x, y

Якщо зазначені в інструкції імена не означені, виникне помилка.

Примітка. У програміста є можливість самостійно викликати збиральник сміття, але слід пам'ятати, що зловживання цими викликами не пришвидшить програму. «Ручне» керування збиранням сміття — не вихід, бо передбачити абсолютно все надскладно і цивілізоване знищення кожного об'єкта гарантовано все одно не буде, а ось погіршити швидкодію — цілком реально. Також більшість використовуваних об'єктів насправді й не потребують цивілізованого знищення. Для решти випадків (виходять за межі даного посібника) набагато краще писати код, який не покладається на те, що об'єкт буде знищено. Обробка числових даних, рядків до таких випадків не відноситься.

Рекомендації з іменування змінних. Синтаксично іменем змінної може бути довільний ідентифікатор, що не є ключовим словом. Хоча ідентифікатори можуть містити літери національних алфавітів, але використовувати цю можливість не рекомендується.

Імена змінних та інших програмних сутностей слід обирати так, щоб вони були пов'язані з тими речами, які вони зображують, наприклад, **temperature**, **radius**, **circle_length** (температура, радіус, довжина кола). Такі імена інколи називають **змістовними**, або **семантичними**. Змістовне ім'я без зайвих коментарів повідомляє, що саме моделює позначувана ним змінна. Використання змістовних імен робить текст програми більш зрозумілим. За класиками: «*Гарна програма має читатися як розмовна мова*».

Імена, означені в бібліотеках системи програмування, використовувати як імена змінних теоретично можливо, але краще цього не робити. Підкреслення першою літерою імені бути може, але такі імена мають спеціальне призначення (буде розглядатися пізніше) і не в усіх ситуаціях можуть бути використані.

В якості імен не варто обирати однолітерні імена **l** (мала латинська *l*), **I**, **O**, оскільки для деяких шрифтів їх доволі легко сплутати з цифрами 1 та 0.

У старі часи, коли довжина імен обмежувалася 8 символами, виникли такі важко читабельні імена, як **clrscr** (отримується з англ. *clear screen* викиданням голосних). Розшифрувати їх можна, але вони все-таки не дуже зручні. Інші системи запису використовують підкреслення (англ. назва такої системи *snake case*), наприклад, **circle_length**. Ще одна дуже поширена система має назву «*верблюжий запис*» (англ. *camel notation*). Якщо ім'я утворюється з кількох слів, то кожне наступне слово записуємо з великої букви. Щодо першої літери, то тут джерела розбігаються: деякі вважають, що перша літера має бути великою, деякі, що малою. Зокрема в PEP8 зазначається, що в системі запису CamelCase (вона ж CapitalizedWords, вона ж CapWords) перша літера верблюжого запису має бути великою; варіант, за якого перша літера мала, названо mixedCase.

Різні частини системної бібліотеки Python розроблялися в різні часи різними людьми та використовують різні стилі іменування змінних. Тим не менш для нових проєктів PEP8 рекомендує для імен змінних використовувати систему запису з використанням підкреслення, записуючи окремі слова з малої букви. Хоча якщо завдяки використанню певних бібліотек у проєкті превалює стиль іменування змінних mixedCase, то заради одноманітності дозволяється використовувати і його.

Якщо потрібна невелика й відома наперед кількість змінних, що позначають однотипні сутності, то в кінці імені можна дописати номер, наприклад, **radius1**, **radius2**.

Існує давня традиція¹⁷, за якою імена **i, j, k, m, n** даються змінним цілих типів. Найчастіше ці змінні позначають індекси (номери) або кількості. Змінні з іменами **x, y, z** найчастіше позначають дійсні (або комплексні) значення.

Змінні, значення яких залишається сталим упродовж виконання програми, тобто змінні, що моделюють константи, можуть позначати великими літерами, наприклад, **CONST**.

5.4 Присвоювання

Первісне знайомство з присвоюванням вже відбулося. За виконання інструкції **target_name = вираз** цільовій змінній **target_name** буде присвоєно значення виразу (за необхідності змінна буде утворена), тобто ім'я **target_name** буде зв'язане з результатом обчислення виразу. У загальному випадку присвоювання може відбуватися не тільки змінній, але й частинам інших змінних, що задаються виразами, але такі випадки зараз не розглядатимуться.

Розглянемо кілька інших випадків присвоювання.

5.4.1 Складені присвоювання

Інструкції присвоювання дуже часто мають вигляд, наприклад, **a = a + b**, **a = a * b** тощо, тобто збільшують значення змінної **a** на величину **b**, множать на **b** тощо. Такі присвоювання можна задати за допомогою так званих *складених*¹⁸ присвоювань, наприклад, **a += b**, **a *= b**, які в одній інструкції комбінують обчислення бінарної операції з подальшим присвоюванням результату обчислення змінній. В якості **b** можна записати будь-який вираз, для якого обчислення виразу (для прикладів відповідними виразами є **a + b** та **a * b**) має сенс. В описаний спосіб до змінної можна застосувати будь-яку з операцій **+**, **-**, *****, **/**, **//**, **%**, ******, а також деякі інші.

Існують випадки, коли виконання складеного присвоювання виконується більш оптимально, оскільки виконується з меншою кількістю проміжних обчислень. Наприклад, замість утворення нового об'єкта та зв'язування його з цільовою змінною (змінна ліворуч від знаку **=**) для деяких типів можливо відразу змінювати значення об'єкта, зв'язаного з цільовою змінною. Якщо присвоювання відбувається не змінній, а частинам інших змінних, що задаються виразами, то за складеного присвоювання кількість обчислень також буде меншою: значення виразу з лівої частини присвоювання буде обчислюватися тільки один раз.

На відміну від звичайного присвоювання, складене присвоювання нову змінну не утворює, але так само як і звичайне змінює значення змінної. Чи буде змінено об'єкт, з яким зв'язано ім'я змінної, залежить від типу змінної. Порядок обчислення цих двох присвоювань теж дещо різниться. За звичайного присвоювання спочатку завжди обчислюється вираз праворуч від знаку **=**, а потім вираз ліворуч. За складеного присвоювання навпаки: спочатку обчислюється вираз ліворуч від знаку **=**, а потім вираз праворуч. У кожній з наведених нижче інструкцій вирази **expr1** та **expr2** обчислюються в порядку, що задано індексами.

```
expr2 = expr1
expr1 += expr2
```

5.4.2 Присвоювання цільовому списку

Присвоювання може виконуватися не тільки одній змінній, але й кільком одночасно.

Присвоювання списку цільових змінних виникає, наприклад, за виконання інструкції такого вигляду:

target_var_1, target_var_2, ..., target_var_N = expr_1, expr_2, ..., expr_N

¹⁷ Ця традиція походить з мови Fortran, в якій типи деяких змінних визначалися за їх іменами.

¹⁸ У документації мови Python використовується термін *augmented*, у деяких інших мовах програмування для таких присвоювань застосовують термін *compound*.

Під час її виконання послідовно обчислюються вирази **expr_1**, **expr_2**, ..., **expr_N**, значення яких далі присвоюються змінним **target_var_1**, **target_var_2**, ..., **target_var_N** відповідно (якщо якоїсь зі змінних не існувало, то вона створюється).

Приклади. Припустимо, що на початку виконання прикладів означено імена **a**, **b** (існують відповідні змінні), а ім'я **c** – ні.

1. За виконання

a, **b**, **c** = 10, 20, 30

буде утворено змінну **c** та змінні **a**, **b**, **c** отримають значення 10, 20, 30 відповідно.

2. За виконання

a, **b** = **b**, **a**

буде виконано обмін значень змінних **a** і **b**. Тепер значеннями змінних **a**, **b**, **c** є значення 20, 10, 30 відповідно.

3. За виконання

a, **b**, **c** = **c**, **a**, **b**

значеннями змінних **a**, **b**, **c** стануть 30, 20, 10 відповідно.

4. За виконання

a, **b**, **a** = **c**, **a**, 15

значеннями змінних **a**, **b**, **c** стануть 15, 30, 10 відповідно.

◀

Слід зазначити, що відомі випадки, коли останнє присвоювання демонструє зовсім не таку поведінку, яка наведена вище. Але, якщо подивитись трохи інакше: чи виникає у повсякденній практиці ситуація, коли в списку цільових змінних імена змінних повторюються? Насправді, така ситуація дуже штучна і гарному коду не притаманна.

5.4.3 Одночасне присвоювання кільком цілям, або Ланцюгові присвоювання

У попередньому випадку різним цільовим змінним присвоювалися різні значення. Присвоювати одне те саме значення різним цілям також можливо. Наприклад, за виконання інструкції

target_var_1 = **target_var_2** = ... = **target_var_N** = **expr**

значення виразу **expr** буде обчислено та присвоєно кожній з цільових змінних **target_var_1**, **target_var_2**, ..., **target_var_N** (при цьому змінні, які до того не існували, будуть створені): кожна з цільових змінних буде зв'язана з одним тим самим об'єктом, який отримується в результаті обчислення виразу **expr**.

Наприклад, за виконання

a = **b** = **c** = 1_000_000_000

під час обчислення літерала 1_000_000_000 буде утворено об'єкт, що зберігає ціле 1000000000, і з цим об'єктом буде зв'язана кожна зі змінних **a**, **b**, **c**.

У випадку виконання послідовності інструкцій

a = 1_000_000_000

b = 1_000_000_000

c = 1_000_000_000

для кожного літерала скоріш за все буде утворений свій окремий об'єкт (незважаючи, що значення однакове) і хоча змінні **a**, **b**, **c** також отримають одне те саме значення, але будуть зв'язані з різними об'єктами.

Зауваження. У мові Python навіть для вбудованих числових типів можлива ситуація, коли існують різні об'єкти з однаковими значеннями. Певна оптимізація може відбуватися, наприклад, для невеликих цілих значень. Доволі ймовірно, що ціле 13 буде зображуватися в пам'яті не більш ніж одним об'єктом. Але документація не рекомендує програмістам покладатися на наявність такої оптимізації.

Зауваження. Для числових типів ланцюгові присвоювання можуть мати сенс. Але у випадку контейнерів необережне використання ланцюгових присвоювань може призводити до неприємних наслідків.

Зауваження. PEP8 рекомендує навколо знаків присвоювань (і звичайних, і складених) ставити пробіли.

5.5 Синтаксична структура програми мовою Python

Програмний проєкт мовою Python може складатися з кількох текстових файлів. Про організацію файлів проєкту на диску мова піде під час розгляду модулів та пакетів.

Якщо розглядати *формальну* специфікацію граматики мови Python, то кожен текстовий файл з кодом мовою Python є послідовністю інструкцій (англ. *statement*), кожна з яких задає певні дії. З теоретичної точки зору інструкція задає перетворення пам'яті програми, тому жодних суперечностей тут не виникає. Розглянемо чому.

Мова Python дозволяє використовувати різні програмні сутності. Ми вже знайомі зі змінними, під час розгляду типів даних кілька слів було сказано про класи, надалі познайомимося з функціями тощо. У Python уся інформація зберігається в об'єктах. Деякі об'єкти утворюються під час створення змінних та зберігають дані у звичному для нас розумінні, деякі об'єкти зберігають службову інформацію про класи, функції тощо. Під час опрацювання тексту програми інтерпретатором за «означеннями», наприклад, класів інтерпретатор створює об'єкти, що зберігають необхідну службову інформацію; при цьому відбувається перетворення пам'яті програми. По інерції інструкції означення доволі часто називають просто означеннями.

Аналогами інструкцій означення в компільованих мовах програмування є означення у звичайному розумінні, які впливають не на поточний стан пам'яті програми, а на виконуваний код, що буде побудовано за вхідним текстом програми.

Проте аналоги багатьох інструкцій мови Python наявні майже в кожній імперативній мові програмування, наприклад, вже знайома нам інструкція присвоювання.

5.6 Вирази, що задають форматування

Раніше зазначалося, що в мові Python наявні f-рядки (починаючи з версії 3.6), що класифікуються документацією як рядкові літерали. Насправді рядкове значення, що його задає f-рядок, утворюється в результаті форматування даних, і по своїй суті f-рядок більше нагадує вираз. У мові Python окрім f-рядків наявні й інші засоби форматування, але зараз буде саме про f-рядки.

Примітка. Python 3.12 вводить деякі уточнення роботи з f-рядками, але ці уточнення пересічні випадки, що розглядаються нижче, не зачіпають.

5.6.1 f-рядки

Ознакою f-рядка є префікс **f** або **F** на початку рядкового літерала. На відміну від звичайних літералів, значення яких задається самим літералом, f-рядки є виразами, що обчислюються під час виконання. Рядковий літерал, перед яким зазначено префікс **f** або **F**, може містити заміщувані поля (англ. *replacement fields*) — вирази у фігурних дужках. Під час обчислення рядкового значення частини, що знаходяться за межами фігурних дужок, залишаються незмінними, замість заміщуваних полів підставляються перетворені на рядки значення виразів. Мова дозволяє задавати спосіб перетворення¹⁹ значення виразу на рядок, форматування та використовувати вкладені заміщувані поля. Щоб задати саме символи **{**, **}** (а не обмежувачі заміщуваного поля) всередині рядкового значення, слід використовувати подвоєння: **{{** та **}}**.

¹⁹ Зараз розглядатися не буде.

Замішчуване поле може мати вигляд **{вираз}** або **{вираз:специфікація формату}**. Якщо специфікація формату не зазначена, вона вважається порожнім рядком.

Для більшості вбудованих типів специфікація формату має відповідати мові специфікацій формату і задає перетворення значення на рядок відповідно до семантики цієї мови. Узагалі ж інтерпретація специфікації формату визначається типом значення, що конвертується.

Опис мови специфікацій формату буде наведено трохи далі. Зараз розглянемо приклади f-рядків на інтуїтивному рівні. Припустимо, що значеннями змінних **n**, **x**, **y**, **w**, **p** є ціле **12**, дійсне **123.4567**, дійсне **-21.89**, ціле **10**, ціле **4** відповідно, та виконано інструкцію **s = 'abc'**. У таблиці 10 наведено f-рядки та рядки, які є їх значеннями.

Табл. 10 — f-рядки та їх значення

f-рядок	Значення f-рядка
f'a{b}c'	a{b}c
	Подвоєні фігурні дужки використовуються, щоб задавати відповідні одинокі фігурні дужки.
f'значення {n} є цілим'	значення 12 є цілим
f'значення {x} є дійсним'	значення 123.4567 є дійсним
f'значення {x = } є дійсним'	значення x = 123.4567 є дійсним
	У Python 3.8 під час наладки стало зручніше виводити сам вираз разом з його значенням.
f'значення {x:12.5f} є дійсним'	значення 123.45670 є дійсним
	Дійсне записано в рядку довжини 12 з 5 знаками після коми у форматі з фіксованою крапкою (f). Оскільки запис числа був коротший 12 символів, то було додано пробіли.
f'значення {x:.3e} є дійсним'	значення 1.235e+02 є дійсним
f'значення {x=:.3e} є дійсним'	значення x=1.235e+02 є дійсним
	Дійсне записано у нормалізованому форматі (e) з 3 знаками після коми. Знаків було більше, тому відбулося округлення.
f'значення {x:.3E} є дійсним'	значення 1.235E+02 є дійсним
f'значення {x:{w}.{p}f} є дійсним'	значення 123.4567 є дійсним
	Дійсне записано у форматі з фіксованою крапкою (f), у 10 символах і з 4 знаками після коми. Загальна кількість символів зображення значення виразу x та кількості знаків після коми (точність) задано заміщуваними полями.
f'{n:+}'	+12
	Символ + у f-рядку каже записувати знак як для від'ємних, так і для додатних чисел.
f'{x:+8.1f}'	+123.5
f'{x:=+8.1f}'	+ 123.5
	Аналогічно попередньому, але символ = у f-рядку каже, що за потреби пробіли слід додавати між знаком та цифрами числа.
f'{n:05}'	00012

f-рядок	Значення f-рядка
	Рядок з десятковим записом доповнюється до 5 символів символом 0.
f'{-n:05}'	-0012
	Якщо в f-рядку довжині результуючого рядка безпосередньо передує символ 0, то за наявності знаку рядок доповнюється нулями між знаком та цифрами числа.
f'{-n: 05}'	-0012
f'{n: 05}'	0012
	Пробіл тут задає форматування знаку: для від'ємних чисел записується -, для додатних — пробіл.
f'{s}'	abc
f'{s=}'	s='abc'

Після того, як за f-рядком вже було утворено рядкове значення, зміна значень змінних, що зустрічаються у виразах заміщуваних полів f-рядка, на отримане рядкове значення вже не впливає.

5.6.2 Мова специфікацій формату

Специфікація формату задає як значення перетворюється на рядок. Вбудовані типи підтримують специфікації такого вигляду.

[[fill]align][sign][#][0][width][grouping_option][.precision][type]

Квадратні дужки позначають, що відповідний елемент може бути відсутній. Деякі опції підтримуються тільки числовими типами, деякі — усіма. Почнемо із загальних опцій.

width — десяткове ціле, що задає мінімальну кількість символів, яку має зайняти перетворене значення, — **ширину поля виведення** значення, — якщо отриманий у результаті перетворення рядок коротший за це значення, то він доповнюється до зазначеної довжини символом-заповнювачем **fill** (може бути довільним символом²⁰) або, за його відсутності, пробілами.

Специфікація дозволяє задати притискання **align**, що визначає спосіб доповнення рядка заповнювачем. Можливі способи притискання зазначено в таблиці 11.

Табл. 11 — Опції притискання

align	Семантика
<	Перетворене значення притискається ліворуч, за необхідності праворуч додаються символи-заповнювачі.
>	Перетворене значення притискається праворуч, за необхідності ліворуч додаються символи-заповнювачі.
^	Перетворене значення центрується посередині поля виведення, за необхідності символи-заповнювачі додаються по обох боках.
=	Ця опція має сенс тільки для числових типів і буде розглянута далі.

За відсутності явних вказівок щодо притискання, для числових типів виконується притискання праворуч, для решти — ліворуч.

precision (точність) — десяткове ціле, що обмежує ширину поля виведення. Для нечислових типів точність задає максимальну кількість символів, яку може зайняти перетворене значення:

²⁰ Символи {та } теж можна використовувати, але напяму записати їх не вдасться.

якщо перетворене значення довше, то зайві символи відкидаються. До перетворення цілих значень вона не застосовна, а для дійсних регулює кількість символів, що виводяться після крапки.

type — символ, що задає спосіб форматування значення, або просто формат. Можливі формати залежать від типу, значення якого перетворюється.

Станом на зараз було розглянуто частини специфікації формату, спільні для всіх типів перетворюваних значень. Подальше вже буде враховувати специфіку.

Перетворення рядка на рядок відбувається тривіальним чином (нема чого перетворювати), тому для конвертації рядків спосіб форматування можна не задавати або вказувати **s**.

Далі розглянемо специфіку саме числових типів.

Для всіх форматів числових типів можна вказувати **sign** – символ, що задає виведення знаку, можливі значення зазначені в таблиці 12.

Табл. 12 — Опції, що задають виведення знаку

sign	Семантика
+	Знак додається як до від’ємних, так і до додатних значень.
-	Знак додається тільки до від’ємних значень.
пробіл	Знак додається тільки до від’ємних значень, для додатних значень замість знаку додається пробіл.

За відсутності явних вказівок знак додається тільки за форматування від’ємних значень.

Формати запису цілих значень. Для цілих значень можна будувати їх десятковий, двійковий, вісімковий або шістнадцятковий запис, чому відповідають формати, що позначаються літерами **d, b, o, x** (та **X**) відповідно. За **x** цифри, що більші 9, записуються малими літерами (a, ..., e), за **X** – великими (A, ..., E).

Згідно документації формат **n** для цілих чисел передбачає утворення десяткового запису із роздільниками груп розрядів відповідно до поточних налаштувань локалізації²¹.

За формату **c** ціле число розглядається як код символу Unicode, і відповідний символ записується в результируючий рядок.

Якщо для цілого значення не вказано спосіб форматування, то за умовчанням використовується формат **d**.

Формати запису дійсних значень типу float. Для дійсних значень можна вказувати способи форматування **e, E, f, F, g, G, n, %**. При цьому якщо точність не задано, то вважається, що вона дорівнює 6.

У форматі **e** дійсне зображується в нормалізованому форматі, між мантисою та порядком вказується літера **e**. Для дійсних типу **float** у **CPython** використовується те саме форматне перетворення, що й у мові C, відповідно до якого порядок записується зі знаком не менш ніж двома цифрами (за потреби на початок дописується **0**); для нуля значення порядку дорівнює нулю. За формату **f** дійсне зображується у форматі з фіксованою крапкою. В обох випадках точність задає кількість знаків після десяткової крапки (якщо дробових розрядів більше, то відбувається округлення). За нульової точності крапка не виводиться.

У форматі **g** дійсне значення **x** залежно від точності зображення може зображуватися як в нормалізованому форматі, так і з фіксованою крапкою. Нехай значення **p** дорівнює:

²¹ Це саме той випадок, коли не треба розуміти все буквально. Які налаштування розуміються як поточні? Налаштування ОС чи налаштування інтерпретатора? Також ніде не сказано, що інтерпретатор за умовчанням підхоплює локалізацію від ОС. Найчастіше за умовчанням транслятори працюють у власній локалізації (прикладом може бути C-locale для компіляторів мов C/C++).

- заданій точності зображення, якщо це значення ненульове;
- **1**, якщо задано нульову точність;
- **6**, якщо точність не задано.

Нехай **exp** – порядок значення *x*. Якщо $-4 \leq \text{exp} < \text{p}$, то використовується формат **f** з точністю **p-1-exp**; інакше використовується формат **e** з точністю **p-1**. В обох випадках незначущі нулі в кінці дробової частини мантиси відкидаються, і, якщо не залишилося жодного дробового розряду, то крапка також відкидається.

У форматах **e**, **f**, **g** для нескінченності та «нечисел» використовуються позначки **inf** (можливо зі знаком) та **nan** відповідно.

Формати **E**, **F**, **G** відрізняються від форматів **e**, **f**, **g** тільки тим, що замість **e**, **inf**, **nan** записують **E**, **INF**, **NAN** відповідно.

У форматі **%** дійсне записується як відсотки: значення множиться на 100 і записується у форматі з фіксованою крапкою, наприкінці додається знак **%**.

Якщо спосіб форматування дійсного значення явно не задано, то за вказаної точності перетворення відбувається аналогічно формату **g**; якщо точність відсутня, то виводиться рівно стільки розрядів, скільки достатньо, щоб відповідний запис за зворотного перетворення на дійсне давав те саме значення, що конвертується, але не менше одного дробового знаку. Наприклад, дійсне **0.1** не може бути зображено точно. Але літерали **0.10000000000000001**, **0.1000000000000000055511151** позначають те саме дійсне число, що й літерал **0.1**. Отже в цьому випадку буде згенеровано один знак після крапки. Для дійсного **10000.** за відсутності формату та точності буде згенеровано рядок **10000.0**. З форматів з фіксованою крапкою та нормалізованим обирається той, який потребує менше символів для запису.

Згідно документації формат **n** для дійсних чисел аналогічний формату **g** з тією різницею, що передбачає утворення десяткового запису із роздільниками груп розрядів відповідно до поточних налаштувань локалізації.

Опція **#** може вказуватися для форматів **b**, **o**, **x** і задає додавання до запису числа префіксу **0b**, **0o**, **0x** відповідно. Для дійсних чисел ця опція забороняє відкидання нулів з кінця дробової частини та дробової крапки (навіть, якщо після неї нема жодної цифри).

grouping_option задає символ-роздільник розрядів (можливими значеннями є **_** та **,**) та може використовуватися з усіма числовими форматами крім **n** та **c**. Для форматів **b**, **o**, **x**, **X** він вставляється між кожними чотирма цифрами, для решти форматів він має семантику роздільника тисяч.

Також для числових форматів (крім **c**) можна вказувати спосіб притискання **=**: за необхідності символ-заповнювач буде додаватися безпосередньо перед цифрами запису числа (але після знаку, якщо він наявний). Той самий ефект буде отримано, якщо записати символ **0** безпосередньо перед шириною поля виведення.

Формати **e**, **f**, **g**, **E**, **F**, **G**, **%** можуть застосовуватися до цілих значень. У такому разі ціле значення попередньо конвертується на дійсне за допомогою **float()**.

Для типу **Decimal** (зберігає дійсні у форматі з фіксованою крапкою, не є вбудованим, наявний у стандартній бібліотеці) можливо застосовувати всі ті форматування, що й для дійсних значень типу **float**. Але при цьому в деяких інтерпретаторах (наприклад, версії 3.7.4) для формату **e** за невказаної точності виводиться не 6, а 5 знаків. У випадку комплексних чисел емпірична перевірка свідчить, що одне те саме форматування застосовується до дійсної та уявної частини як до дійсних значень, отримане рядкове зображення береться в круглі дужки та формат **%** є недоступним.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: об'єкт мови Python, різні об'єкти, тип об'єкта, значення об'єкта, значення зображуваних об'єктом даних, змінюваний об'єкт, типізована мова, статична типізація, динамічна типізація, безтипова мова, змінна в мові Python, зв'язування імені, ініціалізація, присвоювання, сміття, збирання сміття, інструкція **del**, змістовні імена.
2. Які основні характеристики мають об'єкти мови Python?
3. Чи обов'язково об'єкти з однаковим значенням мають бути однаковими?
4. Які характеристики об'єкта можуть змінюватися протягом його існування, а які завжди є незмінними?
5. Чи може під час виконання змінюватися значення, що зображується незмінюваним об'єктом?
6. Чи є змінюваними об'єкти вбудованих числових типів?
7. Які властивості може мати змінна в інформатиці?
8. Які операції можна виконувати над змінними?
9. Якою є типізація мови Python?
10. Які обмеження накладає мова Python на імена змінних?
11. Що відбувається під час зв'язування імені?
12. Які властивості є сталими для змінної мови Python, а які можуть змінюватися під час виконання?
13. Чи правда, що в мові Python змінна ототожнюється з ділянкою пам'яті, де зберігається її значення?
14. Чи може в програмі мовою Python існувати змінна, якій не було надано початкового значення?
15. Що є результатом розіменування змінної?
16. Чи можна розіменовувати неозначені імена?
17. Наведіть приклади виразів, за обчислення яких нове значення не утворюється.
18. Наведіть приклади виразів, за обчислення яких завжди утворюється нове значення.
19. Чи можуть дві різні змінні бути зв'язані з одним тим самим об'єктом?
20. Які засоби керування часом існування змінних мова Python надає програмістам?
21. Чи призводить знищення змінної до негайного вилучення з пам'яті зв'язаного з її іменем об'єкта?
22. Чи завжди об'єкт вивантажується з пам'яті щойно як став невикористовуваним?
23. Що знищує інструкція **del**?
24. З якою метою слід використовувати змістовні імена?
25. Опишіть синтаксис та семантику інструкції присвоювання.
26. Чи можна змінній, що має рядкове значення, присвоїти ціле значення?
27. Як відбувається присвоювання списку змінних?
28. Як записується та виконується складена інструкція присвоювання?
29. Що таке f-рядок і для чого вони можуть бути корисні?

Задачі та вправи

1. Пояснити різницю між **a** і **'a'**.
2. Чому дорівнює значення змінної **x** за виконання такої послідовності інструкцій?
x = 1
x += x

3. Чому дорівнює значення змінної **y** за виконання такої послідовності інструкцій?

x = 1

y = 2.5

y = **x**

x = 3

4. Чому дорівнюють значення змінних **x** та **y** за виконання такої послідовності інструкцій?

x, **y** = 1, 2

x, **y**, **y**, **x** = **y**, **x**, 3, **y**

5. Написати інструкції, за якими створюються змінні **a**, **b**, **c**, з дійсними значеннями 1, 2 та 3 відповідно. Далі записати інструкцію, що задає обмін місцями цих значень «за колом»: значення **a** пересувається до **c**, **c** до **b**, **b** до **a**.

6. Гарантовано, що змінна **ngroup** має ціле значення від 0 до 9. Записати f-рядок, за яким утвориться рядок із шифром відповідної групи першого курсу. Кількість символів за межами заміщуваного поля має бути мінімальною.

7. Нехай змінна **x** має дійсне значення. Записати f-рядок, за яким утвориться рядок із відформатованим значенням змінної **x** згідно таких вимог:

а) використовується формат з фіксованою крапкою, після крапки зазначається рівно 5 знаків;

б) використовується нормалізований формат, після крапки зазначається рівно 2 знаки;

в) використовується нормалізований формат, після крапки зазначається рівно 2 знаки, знак обов'язково виводиться;

г) використовується формат з фіксованою крапкою, після крапки зазначається рівно 2 знаки, знак обов'язково виводиться, ширина поля виводу складає 12 символів, за необхідності між знаком та цифрами додається потрібна кількість пробілів.

8. Змінні **h**, **m**, **s** позначають години, хвилини та секунди відповідно. Записати f-рядок, значенням якого є відповідний час у форматі:

а) **hh:mm** (наприклад, 08:40)

б) **hh:mm.ss** (наприклад, 08:40.23)

9. Змінні **year**, **month**, **day** позначають рік, місяць, день. Записати f-рядок, значенням якого є відповідна дата у форматі:

а) **dd.mm.yy** (наприклад, 01.01.21)

б) **dd/mm/yy** (наприклад, 01/01/21)

в) **dd.mm.yyyy** (наприклад, 17.09.2020)

г) **yyyy-mm-dd** (наприклад, 2020-09-17)

10. Змінна **price** позначає ціну товару в гривнях. Записати f-рядок, значенням якого є рядок вигляду *стільки-то грн стільки-то к.*

Наприклад, за ціни 15.01 має утворюватися рядок **15 грн 1 к.** .

6 Функції

6.1 Підпрограми

Практично в кожній задачі можна виділити окремі частини — *підзадачі*, допоміжні до неї. Якщо програма — це опис розв’язання деякої інформаційної задачі, то підпрограма — опис розв’язання підзадачі.

Деякі підзадачі доводиться розв’язувати в багатьох різних задачах, наприклад, обчислення математичних функцій або введення та виведення даних. Для таких *стандартних підзадач* у кожній системі програмування є величезний набір *готових підпрограм*, зібраних у спеціальні набори — *бібліотеки*. У мові Python підпрограми записуються у вигляді *функцій* (хоча в деяких інших мовах розрізняють функції та процедури), бібліотеки називають *модулями* (англ. *module*), зібрання модулів — *пакетами* (англ. *package*).

У практичному програмуванні знати стандартні підпрограми (або хоча б про їх існування для мови програмування) корисно й необхідно, адже застосовувати готові деталі набагато легше, ніж створювати їх самому. Проте запрограмувати розв’язання «всіх можливих підзадач» — утопія, тому програмістам доводиться постійно створювати власні підпрограми.

Виконання підпрограми (як з стандартної бібліотеки, так й власної) задається її *викликом*, в якому вказано вирази, значення яких мають оброблятися під час її виконання.

Підпрограми розв’язують окремі підзадачі. Це дозволяє:

- краще структурувати код й зробити його більш зрозумілим;
- звести розв’язання складної задачі до розв’язання низки простих; при цьому реалізується загальний принцип «розділяй і пануй», за якого велика задача розбивається на менші, менші розв’язуються окремо і далі з їх розв’язань збирається розв’язання великої задачі;
- уникати дублювання (тобто повторюваних фрагментів) коду;
- повторно використовувати код.

Дублювання коду виникає не тільки тоді, коли в коді наявні посимвольно однакові фрагменти. Воно може мати **приховані форми**, наприклад, коли одні ті самі або аналогічні дії виконуються над змінними з різними іменами або трохи змінюється порядок допоміжних обчислень.

Приклад неявного дублювання коду.

<code>x1 = x + y</code>	<code>tmp = z + y</code>
<code>x2 = y + z</code>	<code>res = cos(tmp) + sin(y + x)</code>
<code>res = sin(x1) + cos(x2)</code>	
<code>sin(x+y*3.5-z)</code>	<code>sin(a+b*3.5-c)</code>

У першому фрагменті коду над змінними *x*, *y* обчислюється один той самий вираз, але обчислення записано трохи по-різному. У другому фрагменті виконується обчислення за однією тією самою формулою, але над різними змінними. В обох випадках маємо дублювання коду, уникнути якого можна за допомогою підпрограм, що задає обчислення виразу. ◀

Дублювання коду погано тим, що:

- 1) при внесенні змін слід не забути внести однакові зміни у всі копії повторюваного коду;
- 2) кожену копію такого коду буде необхідно тестувати (перевіряти) та налагоджувати окремо, тобто витратити більше часу;
- 3) як результат, дублювання коду ускладнює роботу програміста з кодом.

За програмування з підпрограмами ми використовуємо два фундаментальні механізми: *декомпозицію* та *абстракцію*, завдяки яким не тільки покращується структурованість коду та досягається його більша лаконічність, але й побудова коду суттєво полегшується.

Розглянемо приклад з повсякденного життя, щоб показати, як працюють ці механізми.

Візьмемо, наприклад, проектор — чорна (найчастіше) скринька, про внутрішній устрій якої ми майже нічого не знаємо, натомість нам відомий його інтерфейс — кнопка-вимикач, а також є інструкція, в якій написано, як його під'єднати. Якщо його під'єднати до комп'ютера та включити, то він якимось магічним для пересічної людини чином почне відтворювати збільшене зображення на стіну. Ідея абстракції полягає в тім, що не треба знати, як влаштований проектор, щоб мати можливість його використовувати.

Тепер припустимо, що нам треба відтворювати на стіну велике зображення, яке має займати площу більшу, ніж покриває один проектор. У цьому випадку ми можемо розкласти велике зображення на частини і кожен відобразити окремим проектором. Різні пристрої працюють разом, щоб спільно досягти кінцевої мети. Такий підхід викриває суть декомпозиції: велику задачу розкладено на підзадачі, кожен розв'язано окремо, а потім результат зібрано до купи. При цьому завдяки абстракції при збиранні кінцевого продукту нам були важливі не деталі реалізації його частин, а їх зовнішня поведінка. Частини на цьому етапі розглядаються як чорні скриньки, що взаємодіють із зовнішнім світом за певним чітко визначеним протоколом. Ми абстрагуємось від деталей реалізації і для побудови кінцевого продукту використовуємо тільки інтерфейси його частин, які мають бути чітко визначеними. Чому це пришвидшує розробку? Тому, що для досягнення результату ми збираємо результат з невеликої кількості окремих блоків. Це простіше, ніж отримувати результат з нуля.

6.2 Погляд з висоти пташиного польоту на те, як працюють функції

Функція в мові програмування — це частина програми, оформлена в спеціальний спосіб. Використання, або застосування, функції називають **викликом функції**. Синтаксично виклик функції дуже нагадує вирази з функціями, які використовуються в математиці. У тексті програми виклик функції може позначатися виразом, що складається з імені функції, за яким в круглих дужках вказуються вирази, які задають значення аргументів, що мають оброблятися під час її виконання. Кількість аргументів та їх типи у мові Python можуть бути довільні: головне, щоб функція вміла обробляти аргументи в такій кількості і таких типів.

Так, у математиці можна записати вираз **sin(1/7)**, де ім'я **sin** позначає функцію, яка має бути обчислена для її аргументу — виразу **1/7**, при цьому очікується отримати значення функції на цьому аргументі. У програмуванні все майже так само (і ім'я **sin** також позначає функцію), тільки функція вже не щось абстрактне, а конкретна частина програми, тобто код, що виконує певні дії. Також у програмуванні окрім видачі результату своєї роботи функція може виконувати й інші дії, що полягають у зміні значень, що зберігаються в пам'яті програми.

Приклад. Нехай означено функцію з іменем **sin** і виконується інструкція присвоювання.
`x = sin(1/7)`

У процесі її виконання буде обчислюватись значення виразу **sin(1/7)**. Спочатку інтерпретатор обчислить значення виразу **sin** — об'єкт, що задає функцію (а за цим об'єктом і код, який має бути виконаний). Якщо така функція дійсно означена, то далі інтерпретатор перевірить, чи можна її виконати з одним аргументом. У нашому випадку це так, тому далі буде обчислено **аргумент виклику** — значення виразу в дужках, тобто **1/7**. Після результат обчислення (дійсне 0.14285714285714285) разом з керуванням порядком обчислень будуть передані функції **sin** для виконання обчислень. Як тільки обчислення будуть завершені, **виклик функції завершується**: результат обчислення (дійсне 0.14237172979226365) повертається в якості результату обчислення виразу **sin(1/7)**, керування порядком обчислень також буде повернуто в точку виклику, і продовжиться подальше виконання інструкції присвоювання: отриманий результат зв'язується з іменем **x**.

У цьому прикладі можемо дозволити собі дивитись на підпрограму **sin** як на чорну скриньку (див. рис. 11).

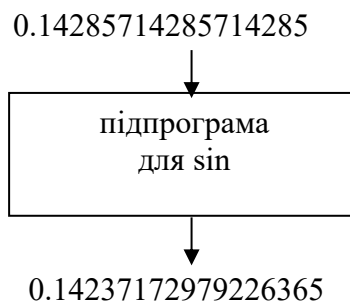


Рис. 11 — Підпрограма для `sin` як чорна скринька

Бібліотеки систем програмування мовою Python містять численні підпрограми, що реалізують математичні та інші функції, наприклад, функцію **print**, яка вже зустрічалася раніше. Функції так званої *стандартної*, або *системної*, *бібліотеки* зазвичай постачаються разом з інтерпретатором. Для їх використання в систему програмування не треба нічого додатково встановлювати. Крім стандартної бібліотеки існують й інші компоненти, які містять функції та додатково можуть бути завантажені з інтернет-ресурсів і встановлені в систему програмування (так звані *сторонні*, або *third-party*, бібліотеки). Інтегрування функцій у систему програмування не є обов'язковим: у програміста завжди існує можливість використовувати довільні сторонні функції, зокрема написати та використовувати власні функції.

Під час виконання код функцій також зберігається в об'єктах. Для того, щоб можна було здійснити виклик функції, її ім'я має бути означено, а код бути доступний інтерпретатору, тобто зберігатися в пам'яті. Стандартна бібліотека містить забагато різних функцій, що використовуються в доволі різних задачах. Малоімовірно, що в якомусь проєкті вони всі будуть одночасно використовуватися. Тому інтерпретатор відразу завантажує тільки доволі невелику кількість функцій та типів, зокрема ті, код яких вбудований у код інтерпретатора (або завантажується під час завантаження інтерпретатора, але це вже технічні подробиці). Для невеликої низки цих *вбудованих* засобів (наприклад, **int**) для їх використання програмісту не треба виконувати жодних додаткових дій: їх код завжди доступний інтерпретатору, і ім'я є означеним. Для використання решти функцій та типів (у тому числі з стандартної бібліотеки) вони мають бути імпортовані з використанням спеціальної інструкції імпортування: за потреби завантажені в пам'ять та їх код має стати доступним для використання.

6.3 Елементи організації консольного введення та виведення

Продовжуючи знайомство з функціями, розглянемо вбудовані бібліотечні функції, що забезпечують консольне введення та виведення даних. Найчастіше виведення даних відбувається на екран, а введені дані зчитуються з клавіатури.

6.3.1 Виведення даних

Виведення даних на екран можна здійснити вбудованою функцією **print**. Ця функція здатна обробити довільну кількість значень довільних типів. Перелік того, що треба вивести, вказується в дужках через кому. Функція **print** за потреби конвертує аргументи виклику в рядки²² (для аргументів, що є рядками, конвертація непотрібна) та виведе їх значення через пробіл у порядку, в якому вони записані в тексті програми. По завершенню виведення аргументів вона виведе

²² Форматування відбувається аналогічно утворенню значення f-рядка у випадку відсутньої специфікації формату.

символ `\n`, що задає перехід на початок нового рядка. За виведення `\n` курсор (поточна позиція виведення) буде переведена на початок нового рядка, і наступне виведення відбуватиметься з початку нового рядка.

За виконання програми

```
print("smth")
print(1)
print(2, 3, 4)
print()
print(5)
```

на консолі побачимо.

```
smth
1
2 3 4

5
```

Зазначимо, що лапки, які задають рядкові літерали в тексті програми, не виводяться: виводиться тільки безпосереднє значення рядка.

Інструкція виразу. Виклик функції є виразом. Зазвичай вирази використовуються як частини інших виразів або в інструкціях, наприклад, в інструкції присвоювання. Але окремо взятий вираз, що не є частиною іншого виразу або інструкції, з точки зору мови також є інструкцією — *інструкцією виразу* (англ. *expression statement*). Семантика інструкції виразу полягає в обчисленні виразу. Під час обчислення виразу можуть відбуватися виклики функцій і виконуватися певні корисні для користувача дії, наприклад, виведення на консоль (але не тільки). В інтерактивному режимі інтерпретатора обчислене значення виразу, що складає інструкцію виразу, може виводитися на консоль. У наведеній програмі усі виклики функцій утворюють інструкції виразу.

Насправді роздільниками аргументів та тим, що виводиться в самому кінці виведення, можна керувати. За виконання програми

```
print(1,2,3, sep=',', end='*')
print(4,5, sep=' delim ')
print(6,7, end=':> ')
print("smth")
print("FINE")
```

на консолі побачимо

```
1,2,3*4 delim 5
6 7:> smth
FINE
```

Порівняно з попереднім прикладом тут з'явилися нові синтаксичні елементи. Спробуємо розібратися.

6.3.2 Позиційні та ключові аргументи виклику функції

У математиці та в більшості мов програмування те, як буде оброблятися аргумент, визначається його позицією в переліку аргументів виклику функції. Наприклад, для математичної функції $f(x, y)$, заданої співвідношенням $f(x, y) = x + 2y$, ми точно знаємо, що значенням $f(1, 2)$ буде 5, а не 6, бо перший аргумент – 1, буде підставлено на місце першого параметра функції, а другий – 2, на місце другого. Якщо б ми запрограмували відповідну функцію мовою Python, то за виклику **`f(1, 2)`** відбувалося б те саме: аргументи підставлялися б у відповідності до позицій у переліку аргументів виклику функції.

Спосіб підстановки аргументів, коли аргумент виклику підставляється згідно своєї позиції, називають *позиційним*, а відповідні аргументи — *позиційними аргументами* (англ. *positional*

arguments). Позиційна підстановка аргументів під час виклику функції притаманна не тільки математиці, але наявна в кожній мові програмування.

Деякі мови програмування, у тому числі Python, крім позиційної підстановки аргументів дозволяють використовувати **ключові**, або **іменовані, аргументи** (англ. *keyword arguments*). Те, як функція буде обробляти ключовий аргумент, визначається не його позицією в переліку аргументів виклику, а його іменем — *ключем* (*keyword*). Проте обробка функцією позиційних аргументів визначається їх позицією в переліку аргументів виклику.

Одночасно у виклику функції можуть бути наявні як позиційні, так і ключові аргументи. При цьому спочатку завжди вказуються позиційні аргументи (за наявності), а потім — ключові (також за наявності). Порядок ключових аргументів між собою значення не має. Імена, які функція може обробити в якості ключів, задаються в означенні функції. Для функцій стандартної бібліотеки про них можна дізнатися з документації.

Ключові аргументи стають корисними, коли існує ймовірність переплутати необхідний порядок аргументів (наприклад, у документах прізвище, ім'я та по батькові можуть зазначатися як в порядку ПІБ, так і в порядку ІПІ; у виклику функції з цими трьома аргументами більш безпечно задавати не позиційні, а ключові аргументи).

Функція **print** виводить свої позиційні аргументи, за допомогою ключових аргументів можна керувати деякими аспектами виведення. В якості ключів вона здатна обробляти імена **sep** та **end** (насправді, це тільки частина переліку). **sep** (є скороченням від англ. *separator* — роздільник) задає який рядок використовувати в якості роздільників аргументів під час виведення; **end** задає який рядок слід вивести по завершенню виведення. Якщо значення цих ключових аргументів у виклику явно не вказано (одного чи обох), то для них використовуються значення **за умовчанням** (англ. *default*): **sep=" "**, **end="\n"**. Як наслідок аргументи виводяться через пробіл, у кінці виводиться **\n**. Зазначимо, що значення параметрів **sep** та **end** можуть бути передані **print** тільки ключовими аргументами.

За використання ключових аргументів програміст може у виклику вказувати тільки ті значення аргументів виклику, що відрізняються від значень за умовчанням. За використання тільки позиційних аргументів виклику це можливо не в усіх випадках. Типовим прикладом тут є **print(6,7, end=':> ')**, де явно вказано значення аргументу **end** виклику функції **print**, але при цьому не вказано значення аргументу **sep**, що передує йому в означенні функції.

6.3.3 Введення значень

Консольне введення в мові Python можна здійснити з використанням вбудованої функції **input**, яка зчитує набрану користувачем на клавіатурі²³ послідовність символів. Ознакою завершення введення є натискання користувачем клавіші **Enter**, за якою генерується символ нового рядка. Отримана від користувача послідовність символів крім заключного символу нового рядка зберігається в рядку типу **str** та повертається в якості результату функції.

За виконання інструкції

```
s = input()
```

комп'ютер зупиняється в очікуванні вхідної послідовності символів. Як тільки користувач набере на клавіатурі послідовність символів, що зображує необхідне йому значення (ці символи з'являться на екрані), та натисне клавішу **Enter**, за введеною послідовністю буде створено відповідний рядок, виклик функції завершиться і отриманий рядок буде присвоєно змінній **s**. Символ **\n** до отриманого рядка при цьому не входить. За послідовного натискання клавіш **A**, **Space**, **B**, **Space**, **C**, **Space**, **Enter**, значенням змінної **s** буде рядок **"A B C"**.

²³ Зараз розглянемо тільки те, як робота цієї функції виглядає ззовні у найбільш звичному випадку.

Увага! На відміну від деяких інших мов пробіли, табуляції (та деякі інші білі символи) з початку/кінця рядка не вилучаються, а введена послідовність символів завершується тільки за появою `\n`.

Явним недоліком щойно записаного коду є те, що комп'ютер зупиняється в очікуванні, і, в реальних програмах, користувач має самостійно здогадуватися, чого саме він очікує. Отримується недружній до користувача інтерфейс. виправимо це. Для цього перед введенням треба повідомити користувачу, що саме від нього очікує програма — вивести *підказку* (англ. *prompt*) на введення. Це можна зробити або раніше розглянутою функцією **print**, або скористатися тим, що запрошення на введення можна передати в якості аргументу функції **input**. В останньому випадку функція почне свою роботу з виведення запрошення, а потім вже буде виконувати дії з введення.

Замінімо інструкцію на таку та натиснемо ту саму послідовність клавіш.

```
s = input("Введіть щось: ")
```

На екрані побачимо

```
Введіть щось: A B C
```

Якщо далі здійснити виведення змінної `s`

```
print(s, end=".\\n")
```

побачимо

```
A B C .
```

Подальше виведення буде виконуватися з нового рядка.

Якщо за виконання виклику функції **input** відразу натиснути клавішу **Enter**, то з виклику буде повернуто порожній рядок.

Функція **input** може приймати або один, або жодного позиційного аргументу. За наявності він розуміється як підказка на введення та має бути рядком²⁴. Значенням, наприклад, числового типу він бути не може.

Введення числових значень. Результатом функції **input** завжди є рядок. З огляду на останнє зупинимось на тому, як задати в програмі введення, наприклад, числових значень. Відповідь доволі очевидна: отриманий за допомогою функції **input** рядок слід явно перетворити на значення відповідного типу. Результатом обчислення виразу *назва_типу(вираз)* є значення виразу, приведене (якщо це можливе) до зазначеного типу.

Розглянемо приклад з введенням цілого числа в змінну `n`. За виконання послідовності інструкцій

```
s = input("input an integer: ")
n = int(s)
```

користувачу буде виведено запрошення на введення цілого числа, далі отримана від нього послідовність символів буде зчитана та записана в рядку `s`, наприкінці значення рядка буде перетворено на ціле та збережено в змінній `n`. Те саме, але без додаткової змінної `s` можна записати так.

```
n = int(input("input an integer: "))
```

Під час перетворення на числове значення білі символи на початку та в кінці рядка відкидаються. Запис вважається десятковим²⁵. У решті правила його запису такі самі, що й для літералів відповідних типів, з наступними винятками:

- запис може починатися зі знаку `+` або `-`;
- ненульове ціле значення може починатися довільною кількістю нулів;
- для типу **float** можна вводити десятковий запис цілого без крапки;

²⁴ Або значенням типу **NoneType**.

²⁵ Для цілих є можливість вказати систему числення, див. документацію.

– для типу **float** можна вводити **infinity**, **inf**, **nan**, у довільному регістрі та зі знаком + або – на початку: **infinity**, **inf** перетворюються на $+\infty$, **nan** – на NaN.

6.4 Бібліотечні математичні функції та константи

Мова Python має низку функцій, що реалізують математичні функції дійсного та комплексного аргументу. Перелік функцій дійсного аргументу дуже близький до переліку функцій аналогічної стандартної бібліотеки мови C (і це не друкарська помилка в поточному електронному документі). Проте певні відмінності все-таки існують: і по переліку, і по нюансам реалізації. Зокрема в реалізації Python поведінка функцій на аргументах, що не належать області визначення, може відрізнятися від поведінки цих функцій згідно стандарту IEEE std. 754 (Standard for Floating-Point Arithmetic) та від поведінки аналогічних функцій мов C, C++. Зокрема, деякі з цих функцій реалізуються безпосередньо в процесорі та за недопустимих аргументів повертають NaN або, коли результат не може бути записаний у 8-байтовому дійсному, повертають нескінченність. У мові Python у таких випадках можуть виникати виняткові ситуації.

Бібліотека математичних функцій дійсного аргументу міститься в модулі **math**. Функції комплексного аргументу зібрані в модулі **cmath**. Зараз розглянемо роботу з математичними функціями дійсного аргументу.

У модулі **math** міститься багато звичних для нас математичних функцій та констант: **sqrt** (квадратний корінь, назва походить від скорочення **square root**), експонента **exp** (основою є число e) та логарифм **log**, тригонометричні функції **sin**, **cos**, **tan** (тангенс), обернені тригонометричні функції **asin**, **acos**, **atan**, гіперболічні та обернені до них функції тощо, а також імена **pi**, **e** задають значення математичних констант π та e .

Серед інших функцій виділимо такі:

- **fabs** повертає абсолютну величину числа;
- **floor**, **trunc** за своїм аргументом повертають відповідно цілу частину та ціле, яке утворюється відкиданням дробових розрядів;
- **gcd** (скорочення від *greatest common divisor*) повертає найбільший спільний дільник своїх цілих аргументів;
- **factorial** обчислює факторіал свого аргументу;
- **pow(x, y)** виконує піднесення x в степінь y , тобто обчислює $x^{**}y$ (**pow** — скорочення від англ. *power* — степінь), результат є дійсним числом;
- **isfinite**, **isinf**, **isnan** виконують відповідні перевірки свого аргументу.

Крім того в модулі означено імена **inf** та **nan**, які позначають відповідні дійсні константи.

Для використання засобів з модуля **math** їх треба імпортувати. Існує декілька синтаксичних способів це зробити. Зараз розглянемо тільки мінімальні відомості, які дозволять розпочати використання математичних функцій.

За виконання інструкції

```
import math
```

усі означені в модулі імена будуть імпортовані та доступні за так званим **кваліфікованим іменем**, наприклад, **math.sin(1.5)** (неформально запис **math.sin** дослівно можна прочитати так **sin** з **math**).

За виконання інструкції

```
from math import sin
```

з модуля **math** буде імпортовано тільки означення функції **sin**, до якої можна буде звертатись за простим ім'ям, наприклад, **sin(1.5)**.

За виконання інструкції

```
from cmath import sin as c_sin
```

з модуля **cmath** буде імпортовано тільки означення функції **sin**, до якої можна буде звертатись за іменем **c_sin**, що може бути корисним, якщо одночасно треба працювати як з дійсними, так і з комплексними версіями математичних функцій.

Приклад. За виконання програми

```
from math import pi, e, cos

print("e = ", e)
print("pi = ", pi)
print("cos(pi) = ", cos(pi))
```

буде виведено

```
e = 2.718281828459045
pi = 3.141592653589793
cos(pi) = -1.0
```

6.5 Написання та використання власних функцій

Напишемо функцію, що за заданими значеннями x та y обчислює значення виразу $x + y$.

```
def first(x, y):
    tmp = x + y
    return tmp
```

У прикладі на початку функції записано **заголовок** функції. Заголовок функції починається з зарезервованого слова **def** (від англ. *definition* — означення); далі записується ідентифікатор — **ім'я** функції (у прикладі це **first**), далі в круглих дужках записується перелік **формальних параметрів** — змінних, що під час виконання виклику будуть зв'язані зі значеннями-об'єктами, над якими мають бути виконані обчислення, — **аргументами виклику** функції, або **фактичними параметрами**. Це дає можливість застосовувати функцію для обчислення над різними значеннями. Для кожного формального параметру вказується його ім'я, яке має бути ідентифікатором. Якщо формальних параметрів декілька, між ними ставиться кома. Далі записано **тіло** функції, що синтаксично є **набором інструкцій** (у термінології документації це *suite*) — оформленою в спеціальний спосіб послідовністю інструкцій, і задає виконувани функцією дії.

У прикладі тіло функції складається з двох інструкцій. За першою виконуються обчислення і результат зберігається в змінній **tmp**, що створюється. За другою — інструкцією **return**, — виконання функції завершується й значення виразу записаного після ключового слова **return** повертається як результат виклику функції.

Зауваження. У Python функції прийнято іменувати аналогічно змінним, тобто з використанням snake_case, наприклад: **find_root**.

6.5.1 Прості та складені інструкції, набори інструкцій (suite)

Інструкції мови Python можна поділити на **прості** (*simple*) та **складені** (*compound*). Прості інструкції записуються не більш ніж на одному логічному рядку. Прикладами простих інструкцій є присвоювання, складені присвоювання, інструкція **return**, інструкція виразу. Крім зазначених простих інструкцій існують й інші, з якими будемо поступово знайомитися.

Складені інструкції містять у собі групу інших, вкладених, інструкцій. Вони складаються з однієї або кількох частин (англ. *clause*), кожна з яких має свій заголовок (англ. *header*) та підпорядкований йому набір інструкцій (англ. *suite*).

Заголовок складеної інструкції майже завжди починається із зарезервованого слова та завершується двокрапкою.

Набір інструкцій це

– або непорожня послідовність простих інструкцій, записаних через ; на одному логічному рядку (можливо не від початку, але до кінця), наприклад,

```
x = 1; y = 1 # приклад набору інструкцій
x = 2       # вже інший набір інструкцій
x = 4       # ще один набір інструкцій
```

після набору інструкції, записаного в один рядок завжди має йти лексема **NEWLINE** (ознака завершення логічного рядка).

– або послідовність складених інструкцій та однорядкових наборів інструкцій, що синтаксично оформлюється з нового рядка та записується із збільшенням абзацного відступу; у цьому випадку вона має вигляд.

NEWLINE INDENT *послідовність складених інструкцій і однорядкових наборів інструкцій*
DEDENT

Прикладом такого набору є тіло функції у прикладі вище.

6.5.2 Поняття блоку в програмуванні

Доволі часто можна побачити, що набір інструкцій (*suite*) називають блоком (англ. *block*). Не намагайтеся знайти істину в цьому питанні. Це вкрай шкідливо для здоров'я. Бо документація Python каже одне, а програмістська спільнота, яка володіє не тільки мовою Python, дещо інше, прив'язане до інших мов програмування. І кожна сторона має рацію.

Термін «складена інструкція» в Python має дещо інший зміст, ніж у багатьох інших мовах програмування. Якщо в Python у цю категорію потрапляє певна сукупність інструкцій, то в мовах C, C++ складена інструкція — це цілком конкретна інструкція мови, що є послідовністю інструкцій, оформленою в спеціальний спосіб (навколо ставлять операторні дужки { }). Також у тих мовах складену інструкцію називають блоком (англ. *block*). Саме блок має бути тілом функції у мовах C, C++. Цю аналогію переносять на мову Python і підміняють термін *suite* на термін *block*.

У документації на мову Python одночасно існують і *suite*, і *block*. Під *suite* розуміється оформлена в спеціальний спосіб послідовність інструкцій. Одночасно з цим введено поняття блоку (англ. *block*) — фрагменту коду як виконуваної одиниці тексту (блок відноситься до внутрішньої кухні інтерпретатора). Правда полягає в тім, що відповідність між наборами інструкцій (*suite*) тексту та блоками виконуваного коду (*block*) не є взаємно однозначною. Блок з точки зору інтерпретатора Python це може бути дещо більше, ніж один набір інструкцій.

З точки зору **парадигми структурного програмування** кожна програма має зображуватися ієрархічною структурою блоків. У цьому випадку під **блоками** розуміють логічно згруповані послідовності інструкцій, які можна розглядати як єдине ціле, зокрема, звертатися до них як до однієї інструкції. Зокрема, тіло функції можна розглядати як блок. І в цьому сенсі набір інструкцій блоком є.

Висновки:

- 1) *suite* варто сприймати як синтаксичний елемент з опису мови;
- 2) треба знати, який вигляд може мати *suite*, бо без цього будуть проблеми з написанням коду;
- 3) переклад та синоніми, які для *suite* можна побачити в інтернеті, — це другорядне.

6.5.3 Інструкція означення функції та інструкція *pass*

Інструкція означення функції є складеною інструкцією, що в найпростішому випадку має вигляд

def ім'я_функції (перелік_формальних_параметрів): набір_інструкцій

За її виконання записані в тілі функції інструкції не виконуються. Натомість інтерпретатор створює об'єкт, що містить код функції та певну службову інформацію, та зв'язує цей об'єкт з іменем функції.

Інструкція означення функції може розташовуватися серед інших інструкцій. Не обов'язково на початку файлу. Єдине синтаксичне обмеження, яке накладається: на момент використання функція має бути вже означена. Також мова Python підтримує вкладені функції: у тілі функції дозволено означати інші функції²⁶. Незважаючи на таку свободу вибору місця для запису функції, записувати їх всуміш з іншими інструкціями не варто. Зазвичай функції записують на початку файлу перед кодом, що їх використовує. Також не слід допоміжні для розв'язання деякої задачі функції оформлювати як вкладені в основну функцію. Призначення вкладених функцій у мові Python зовсім інше і пов'язане з аспектами функціонального програмування.

Якщо треба відтермінувати в часі реалізацію якоїсь функції (запис виконуваних нею дій), виникає бажання записати порожній набір інструкцій. Зробити це забороняє синтаксис. У такій ситуації можна скористатися інструкцією **pass** (можна перекласти як *пропустити*), що не задає жодних дій. Наприклад.

```
def bad_function(): pass
```

Тепер ця функція формально має тіло (що є однорядковим набором інструкцій) і не буде заважати тестувати інші функції. Зрозуміло, що згодом вона має бути належно реалізована: відповідний код має бути записаний замість **pass**.

Інструкція **pass** буває доречна і в інших випадках, коли жодних дій виконувати не треба, але записати хоч одну інструкцію треба.

6.5.4 Приклади використання функцій у програмі

Розглянемо програму, що містить означення функції та її численні виклики.

У програмі означено функцію **first**, змінні **x**, **y**, далі записано інструкції присвоювання з викликами функції **first** та виведення значення змінної **res** на консоль.

У тексті програми виклик функції може позначатися виразом, що складається з імені функції, за яким в круглих дужках вказуються вирази, що задають значення аргументів, що мають оброблятися під час її виконання, наприклад, **first(4.0+1.0, -2.0)**, **first(x+y, y-1)**, **first(y, x)** тощо.

²⁶ Не все, що дозволено використовувати, варто використовувати. У більшості випадків замість вкладених функцій має сенс розглянути можливість організувати відповідний код у вигляді класу та його методів. Крім того, вкладені функції підтримуються не всіма мовами програмування; якщо надалі треба буде перенести код на іншу мову, то за відсутності засобів автоматизації вкладені функції цей процес ускладнять. Випадки, коли вкладені функції дійсно корисні, доволі специфічні.

```

def first(x, y):
    tmp = x + y
    return tmp

x, y = 1.0, 2.0

res = first(0.0, 0.0)
print(res)

res = first(4.0+1.0, -2.0)
print(res)

res = first(x+y, y-1)
print(res)

res = first(1, 0.0)
print(res)

res = first(6, 7)
print(res)

res = first(x, y)
print(res)

res = first(y, x)
print(res)

```

У першому випадку обчислення буде проводитися над двома дійсними значеннями **0.0, 0.0**, тобто вони стануть аргументами виклику, і далі буде виведено **0.0**; у другому випадку аргументами виклику будуть значення **5.0, -2.0** і далі буде виведено **3.0**; за третього виклику аргументами будуть **3.0, 1.0** і далі буде виведено **4.0**; у четвертому виклику аргументами будуть ціле **1** та дійсне **0.0**, під час обчислення $+$ ціле значення **1** буде *неявно* приведенне до дійсного, функція поверне дійсне значення і далі буде виведено **1.0**; у п'ятому виклику аргументами виклику будуть цілі **6, 7**, обчислення будуть виконуватися над цілими, функція поверне ціле **13**, яке й буде виведено. З кожного з двох останніх викликів функція поверне дійсне **3.0**. ◀

Аргументами функції можуть бути значення довільного типу: головне, щоб над ними могли бути виконані зазначені в тілі функції дії (якщо це буде не так, то виникне виняткова ситуація і, якщо вона не буде належним чином оброблена, то виконання програми буде аварійно завершено). Одночасно з цим за кількістю перелік аргументів виклику має відповідати переліку формальних параметрів функції (якщо явно не зазначено інше).

Тип результату функції також не регламентується. Він може бути фіксованим, може залежати від типів аргументів або навіть функція за різних гілок обчислень може повертати значення різних типів.

Щодо останніх двох викликів функції **first** у прикладі, то імена змінних, що задають аргументи виклику, не мають жодного відношення до імен формальних параметрів функції (однофамільці ☺). Під час виклику функція отримує значення, які, неформально кажучи, підставляються на місця формальних параметрів.

У наведеному прикладі використовувалися виклики функції з позиційними аргументами. Запишемо приклади викликів, де наявні ключові аргументи.

Виклик **first(1, 3)** з використанням ключових аргументів можна записати кількома способами: **first(x=1, y=3)**, або **first(y=3, x=1)**, або **first(1, y=3)**.

Запис **first(3, x=1)** є неправильним. Спочатку, за наявності, позиційні аргументи зв'язуються з відповідними формальними параметрами, далі підставляються ключові аргументи. У прикладі формальний параметр **x** отримує два різних аргументи (один позиційний, інший ключовий) — і це є помилкою, а параметр **y** не отримує жодного аргументу — і це також помилка. **Під час виклику кожен формальний параметр має бути зв'язаний рівно з одним аргументом виклику.**

6.5.5 Завершення виконання виклику функції

Завершення виконання виклику функції може відбуватися за інструкцією **return** або, коли завершиться виконання останньої інструкції її тіла.

За інструкцією **return вираз** обчислюється значення виразу, воно стає результатом, який повертається з виклику функції, виконання виклику функції завершується і далі продовжує виконуватися код, з якого було здійснено виклик.

Інструкція **return** може бути записана без зазначення виразу, значення якого слід повернути, наприклад.

```
def second():  
    print("second")  
    return  
    print("This text won't be printed.")
```

За виконання виклику **second()** буде виведено рядок

```
second
```

і далі виконання виклику функції буде завершено (повідомлення **This text won't be printed.** виведено не буде).

Постає природне питання: що буде повернуто з виклику? Аналогічне питання постає стосовно виклику функції **third()**.

```
def third():  
    print("third")
```

Під час виконання виклику виконується остання інструкція її тіла і також відбувається повернення з функції.

У мові Python наявний вбудований тип **NoneType**. Цей тип містить рівно одне значення, що позначається ключовим словом **None**. Якщо значення, що має повертатися з функції, явно не зазначено, то повертається саме значення **None**.

Інструкцію **return** можна використовувати і в такому синтаксисі:

return послідовність_виразів_через_кому

Не вдаючись в синтаксичні деталі, на прикладі покажемо, як це може працювати. За виконання коду

```
def f():  
    return 1, 2  
  
x, y = f()
```

з виклику **f()** буде повернуто кортеж з двох значень, в результаті присвоєння перше буде присвоєно змінній **x**, друге — змінній **y**.

6.6 Локальні змінні функції

Можлива ситуація, коли одні ті самі імена в різних контекстах позначають різні змінні. Модифікуємо нашу програму.

```

def fourth(x, y):      #0
    x *= 2             #1
    y *= 3             #2
    tmp = x + y        #3
    return tmp         #4

x, y = 1.0, 2.0        #5

res = fourth(x, y)     #6
print(x, y, res)       #7
res = fourth(1, 3)     #8

```

За її виконання буде виведено.

```
1.0 2.0 8.0
```

Можна бачити, що значення змінних **x** і **y**, створених за виконання рядка (5), не змінилися. Під час виклику **fourth(x, y)** функція отримує об'єкти, що є значеннями відповідних змінних. Отримані значення зв'язуються з іменами формальних параметрів і утворюють **нові** змінні, які існують тільки впродовж виконання виклику і в тілі функції позначаються іменами **x**, **y** її формальних параметрів. Ці змінні відносять до локальних змінних функції.

За виконання присвоювання **tmp = x + y** буде утворена змінна **tmp**, яка теж буде локальною змінною функції. Як тільки виконання виклику завершиться, локальна змінна **tmp** припинить своє існування. За наступного виклику змінна **tmp** так само розпочне своє існування тільки після виконання рядка (3) коду: у рядках (1) та (2) розіменовувати ім'я **tmp** не можна, бо відповідне ім'я ще не означене.

- **Локальними** змінними функції є змінні, утворені під час виконання її виклику, а також змінні, що відповідають формальним параметрам функції. Локальні змінні функції припиняють своє існування із завершенням виклику функції та не зберігають своє значення між викликами.
- В якості імен локальних змінних можуть використовуватись імена, які були означені за межами функції. У такому випадку під час виконання виклику зовнішнє відносно функції означення тимчасово припиняє свою дію (але його дані з пам'яті не вивантажуються) і відповідне ім'я розглядається як локальне означене під час виклику ім'я.
- Означені за межами функції імена є глобальними, відповідні змінні називають **глобальними**. Але функції до глобальних змінних відносити не прийнято, хоч відповідні імена й є глобальними і інтерпретатор в однаковий спосіб опрацьовує і імена звичайних змінних, і імена функцій.

Використання глобальних змінних, утворених у результаті присвоювання, у функціях є вкрай небажаною програмістською практикою, оскільки не завжди гарно ладнає з багатопоточністю, може призводити до помилок за неуважністю і ускладнює тестування та наладку коду. Тим не менш існують окремі випадки, коли використання глобальних змінних є доцільним. Мова Python допускає використання глобальних змінних у функціях, що буде розглянуто пізніше.

Розглянемо ще один приклад.

```

def f1():
    f2()

def f2(): pass

f1()

```

Жодних помилок у ньому нема. Імена **f1** та **f2** тут є глобальними, але позначають не змінні, а функції. І ми маємо право в тілі однієї функції викликати іншу. Більш того, не викликаючи з одних функцій інших, побудова великих програмних комплексів була б просто неможливою.

Щодо порядку розташування в тексті, то мова Python не передбачає попереднє оголошення функцій. Взаємне розташування функцій **f1**, **f2** може бути довільним. Головне, щоб на момент виконання виклику, відповідна функція вже існувала (ім'я було означене).

6.7 Тільки позиційні та тільки ключові параметри

Розглянемо функцію, що має два формальні параметри.

```
def f1(a, b): pass
```

У виразі, що задає виклик функції **f1**, ми можемо вказати або два ключові аргументи, або два позиційні, або перший позиційний, а другий ключовий.

Новий синтаксис, що з'явився в Python 3.8, дозволяє обмежувати можливості використання позиційних та ключових аргументів.

Запишемо функцію **f2** з використанням зазначених синтаксичних можливостей.

```
def f2(a, /, b, *, c, d): pass
```

У переліку формальних параметрів з'явилися два нових елементи: знаки **/** та *****. Ці знаки виконують функцію роздільників: усі параметри, записані до знаку **/** (якщо він наявний), є тільки позиційними (англ. *positional-only*): мають передаватися функції тільки як позиційні аргументи; усі параметри, записані після знаку ***** (якщо він наявний), є тільки ключовими (англ. *keyword-only*): мають передаватися функції тільки як ключові аргументи; решта параметрів можуть передаватися і як позиційні, і як ключові аргументи.

Так, у наведеному прикладі, параметр **a** має передаватися тільки як позиційний аргумент, параметр **b** може передаватися і як позиційний, і як ключовий, параметри **c** та **d** можуть передаватися тільки як ключові.

Роздільники **/** та ***** у переліку формальних аргументів не є обов'язковими: їх може не бути взагалі або може бути присутнім тільки один з них.

Тільки позиційні параметри можуть бути корисні у випадках, коли ім'я параметра не має смислового навантаження або коли функція ще знаходиться в розробці та ім'я параметра ще не є зафіксованим.

Тільки ключові параметри можуть бути корисні у випадках, коли використання позиційних аргументів може призводити до плутанини, наприклад, день-місяць-рік або прізвище-ім'я-по-батькові краще завжди передавати тільки як ключові аргументи.

6.8 Значення за умовчанням для параметрів функції

Припустимо, що ми вирішили написати функцію, що виводить дійсне число у форматі з фіксованою крапкою. Далі з'ясувалося, що найчастіше має виводитися 4 знаки після крапки. Але в особливих ситуаціях, наприклад, 10 знаків. Якщо написати функцію, що виводить саме з 4 знаками після крапки, то для особливих ситуацій треба буде писати іншу функцію. Виникне дублювання коду, що погано.

Найбільш розумним виходом буде передавати необхідну кількість знаків як аргумент. Але тоді у звичайних ситуаціях треба буде постійно передавати «стандартне» значення — 4. По-перше, прийдеться набирати кілька зайвих літер. По-друге, підставляти кожного разу неіменоване значення 4 теж погано (якщо надалі уподобання зміняться, то важко навіть уявити в скількох точках коду треба буде виконати зміни).

Щоб вирішити ці проблеми, у заголовку функції для відповідного параметра можна вказати **значення за умовчанням** (англ. *default argument*). Якщо у виклику функції задано значення не для всіх її формальних параметрів, то замість відсутніх значень використовуються відповідні значення за умовчанням. Функція може мати довільну кількість параметрів, для яких вказано

значення за умовчанням. Але всі вони повинні розташовуватися в кінці переліку формальних параметрів.

Приклад. За виконання програми

```
def float2str(x, precision=4):  
    return f'{x:.{precision}f}'  
  
print(float2str(13.5))  
print(float2str(13.5, 6))
```

буде виведено

```
13.5000  
13.500000
```

Для аргументу **precision** функції **float2str** вказано значення за умовчанням – **4**. Тому у виклику **float2str(13.5)** в якості **precision** використовується «стандартне» значення **4**. Під час виконання виклику **float2str(13.5, 6)** значенням **precision** є аргумент виклику, тобто **6**. ◀

Важливі особливості. Значення за умовчанням обчислюються один раз під час обробки інтерпретатором означення функції і отриманий об'єкт далі зберігається в пам'яті інтерпретатора (інтерпретатор далі має до нього доступ через об'єкт, що зберігає функцію). Для вбудованих числових типів та рядків це наслідків не має; але існують й інші типи, для яких про цю особливість варто пам'ятати.

6.9 Перевантаження функцій

Оголошення різних сутностей програми з тим самим ім'ям, що одночасно доступні в одному тому самому фрагменті коду, називається **перевантаженням** (англ. *overloading*) **імені** (інколи, не дуже точно, перекладають як *перезначення імені*).

Виходячи з семантики змінних та означення функцій, мова Python цей механізм не підтримує. Кілька функцій з одним тим самим ім'ям, але різними наборами формальних параметрів, одночасно існувати **не можуть**. Щойно ми запишемо іншу функцію з тим самим ім'ям, буде створено відповідний об'єкт та зв'язано з іменем функції. Доступ до попередньої реалізації функції за цим іменем буде втрачено. Більш того, якщо означити функцію, однойменну з деякою змінною, то надалі відповідне ім'я буде позначати функцію, а не змінну. Також змінній, що позначає функцію, можна присвоїти інше значення: не тільки функцію, але й якесь числове значення. В останньому випадку відповідне ім'я вже не буде позначати функцію.

Одночасно з цим завдяки динамічній типізації одна та сама функція може викликатися з аргументами різних типів. Також є механізми, які дозволяють записувати функції, що, як здається ззовні, можуть приймати різні кількості аргументів.

Приклад.

```

f = 10
# ім'я f зараз позначає цілу змінну

def f():
    print(13)
# тепер f позначає функцію, що виводить ціле 13

def f():
    print(23)
# тепер f позначає функцію, що виводить ціле 23;
# попередня функція недоступна

f()
# у результаті виклику буде виведено 23

f = 77
# тепер ім'я f позначає цілу змінну

```

6.10 Способи передачі аргументів у функцію – як це буває в інших мовах програмування

У програмуванні розрізняють два основні способи передачі аргументів у функцію під час її виклику: *за значенням* (англ. *by value*) та *за посиланням* (англ. *by reference*).

За передачі за значенням аргумент копіюється в пам'ять виклику; подальші дії виконуються тільки над копією. У випадку передачі за посиланням значення не копіюється, натомість формальний параметр ототожнюється з аргументом виклику: усі дії, що відбуваються над формальним параметром, насправді відбуваються над аргументом виклику. Передачу за посиланням використовують тоді, коли функція має змінювати значення змінних.

Розглянемо приклад. Припустимо є дві функції: з іменами **f** та **g**, кожна з яких має формальний параметр **x**, і під час свого виконання збільшує його на 1. Нехай у програмі існує змінна **y**, значення якої дорівнює 1. Нехай функція **f** отримує аргумент за значенням, а функція **g** — за посиланням.

Тоді за виклику **f(y)** значення змінної **y** не зміниться: усі зміни будуть відбуватися тільки над копією її значення. Але за виклику **g(y)** значення змінної **y** збільшиться на 1: у функції виконуватимуться перетворення саме ділянки пам'яті, де зберігається змінна **y**, копія її значення утворюватися не буде.

Обидва ці терміни виникли за часів, коли змінну розуміли як іменовану ділянку пам'яті. У мові Python усе складніше: ділянка пам'яті, де зберігається значення, змінною не є; а зміна значення змінної доволі часто є зв'язуванням імені змінної з іншим об'єктом (ділянкою пам'яті). Ні, якщо зануритися в технічні подробиці реалізації, то ділянку пам'яті знайти можна (десь інформація про зв'язок імені та об'єкта все-таки зберігається), але це дуже філософськи неправильний підхід.

Якщо відштовхуватися від того, що по суті змінна є зв'язком імені з об'єктом, то передача аргументів у Python відбувається виключно за значенням: передається не сам зв'язок, а результат розіменування, тобто значення зв'язку. Якщо забути про технічні подробиці і виходити з точки зору того, що значення аргументу виклику все-таки не копіюється, а віддається оригінал — сам об'єкт (фактично ділянка пам'яті, де зберігається значення), що є значенням, то це відповідає передачі посилання на значення.

У книжках та в інтернеті можна побачити багато цікавого на цю тему. Аж до того, що числа передаються за значенням, контейнери — за посиланням. Насправді, все простіше: обговорювані терміни передбачають дещо іншу модель змінної, ніж її реалізує мова Python.

6.11 Документування коду

З точки зору синтаксису коментар не є обов'язковим елементом програми, але з точки зору гарного стилю програмування дуже бажано повідомити призначення змінних та обмеження на їх значення (особливо це стосується глобальних змінних та параметрів функцій), призначення програмного модуля (текстового файлу з кодом), призначення та особливості функції, тобто **задокументувати** написаний код.

Гарна документація дозволяє використовувати код, не вникаючи в подробиці його реалізації. Наприклад, для функції

```
def f(x, y):
    # Return quotient of arguments.
    # pre: y should be nonzero.
    #      Don't check if y is nonzero. If not so, raise an exception.
    tmp = x / y
    return tmp
```

відразу після заголовка зазначено її призначення, вимоги до аргументів та наслідки їх недотримання.

У наведеному прикладі використано коментар, і це дуже нетипово для мови Python, бо в ній є дещо інші механізми документування.

Для функції найчастіше документують:

- призначення функції, тобто що саме вона виконує;
- яким є зміст її параметрів, зокрема, вказуючи ті, що можуть оброблятися як ключові;
- які існують обмеження на типи її аргументів;
- результат якого типу вона повертає;
- **передумови** (англ. *preconditions*) — умови, які мають виконуватися *перед* її викликом, зокрема, умови для її аргументів (фактичних параметрів);
- **післяумови** (англ. *postconditions*) — умови, що виконуються *після* її виклику, зокрема, як функція змінює глобальні змінні програми та значення аргументів (якщо це можливо);
- поведінку функції за некоректних фактичних параметрів.

Узагалі, **наявність вдалої документації на функцію спрощує її подальше використання** в програмі: щоб дізнатися як правильно використати функцію достатньо тільки прочитати коротку документацію на неї; розбиратися в її коді (який може бути доволі складним) вже не потрібно.

Пам'ятайте: функція документується не для інтерпретатора, а для програміста, який має її використовувати. Отже, коментарі та заголовки слід писати так, щоб, не читаючи тіла функції, можна було зрозуміти, як нею користуватися, які обмеження на використання і які наслідки порушення обмежень.

Кілька слів про стиль. Стиль запису коментарів, як і весь стиль проєкту може визначатися не тільки різними PEP, але й політикою організації та іншими чинниками. Зважаючи на те, що сьогодні розповсюджені інтернаціональні команди розробників, коментарі прийнято записувати англійською мовою. Крім того, коментарі хоч і рекомендують записувати повними реченнями, але при цьому їх прийнято записувати в імперативі (наприклад, Return smth.) або в Present Simple (наприклад, Function returns smth.). Слід зазначити, що деякі приклади в The Python Tutorial іноді порушують рекомендації з PEP.

У Python код документують, комбінуючи **рядки документації** (англ. *docstrings*), **анотації** (англ. *annotations*) та звичайні коментарі. Рядки документації та аотації доступні під час

виконання. Коментарі переважно описують більш дрібні моменти та під час виконання не доступні.

Рядки документації. Рядковий літерал, що є першою інструкцією модуля, функції або класу, розглядається інтерпретатором як *рядок документації*. PEP 257 рекомендує використовувати трикратні подвійні лапки, а також записувати рядок документації до модуля та до усіх означених у ньому функцій, класів, призначених для використання ззовні модуля²⁷ (тобто того, що складає інтерфейс модуля).

Рядок документації може записуватися на одному рядку або на кількох. У випадку запису на кількох рядках на першому рядку кратко записують призначення і наступний рядок залишають порожнім. Закриваючі лапки прийнято розташовувати на окремому рядку. Використовувати *f*-рядок в якості рядка документації не можна.

У документації до модуля в цілому вказують його призначення, перелік глобальних змінних, підтримувані параметри командного рядка.

У документації до функції вказують її поведінку, документують призначення аргументів, побічний ефект, обмеження та помилки (винятки), що можуть виникати за її використання.

Анотації призначені для того, щоб задокументувати очікуваний тип значення параметра функції або його призначення, тип результату функції або тип змінної (за винятком локальних змінних функцій).

Анотація задається виразом, значення якого може приймати довільний тип. Вираз, що анотує, може задавати тип, бути рядком з описом параметра тощо. **Інтерпретатор відповідність анотації не перевіряє.** Тим не менш анотації можуть бути дуже корисними за використання сторонніх засобів, що виконують перевірку типів або автоматичне документування коду. Також на підставі анотацій текстовий редактор середовища розробки може давати підказки програмісту.

Якщо в проєкті не використовуються спеціалізовані засоби, то в якості анотацій більш доцільно використовувати типи.

Синтаксично анотація змінної може мати один з двох наведених виглядів:

ім'я змінної: вираз, що задає анотацію = вираз, що задає значення

ім'я змінної: вираз, що задає анотацію

Такий самий вигляд мають і анотації формальних параметрів функції. Анотація результату функції записується в заголовку, за неї заголовок набуває вигляду.

def ім'я функції (перелік формальних параметрів) -> вираз, що задає анотацію:

Якщо типи аргументів та результату функції можуть бути описані за допомогою анотацій, то слід надати перевагу саме такому способу. Включати ці відомості до рядку документації чи коментарів у такому разі не слід.

Доступ до рядків документації та анотацій. І модулі, і функції у мові Python зберігаються як об'єкти вбудованих службових класів, що обслуговують різні синтаксичні елементи програми. Члени класів у документації до мови Python доволі часто називаються *атрибутами*. Цей термін переносять і на об'єкти, що зберігають значення типів, що є класами. Значення виразу **var.member** обчислюється приблизно так: спочатку розіменовується ім'я **var** і отримується деякий об'єкт, далі для отриманого значення шукається значення його атрибуту **member**.

Рядки документації зберігаються в атрибуті **__doc__** відповідного об'єкта; анотації — в атрибуті **__annotations__**.

Для виконуваної програми рядок документації та анотації її змінних доступні за простими іменами **__doc__** та **__annotations__**. Для функції **f** доступ до відповідної інформації можна

²⁷ До глобальних змінних також рекомендується записувати рядки документації. Але деякі інтерпретатори обробляють їх доволі дивно.

отримати через так звані кваліфіковані вирази **f.__doc__** та **f.__annotations__**. За виклику **help(f)** буде виведено рядок, що серед іншого містить рядок документації та анотацію функції. Доступ до рядків документації надається тільки на читання.

Рядки документації функцій обчислюються інтерпретатором під час обробки означення функції і стають доступними незалежно від того, чи здійснювалися виклики функції. У коді тіла функції вони не зберігаються (і, як наслідок, під час викликів повторно не обчислюються).

Приклад документування. За виконання програми

```
"""
This module is designed to demonstrate module and function documenting.
"""

def f(x: int, y: int = 1 * 2) -> int:
    """
    Return the remainder.

    Keyword arguments:
    x - dividend
    y - divider (default 2)

    pre: y!=0
        If not so, raise ZeroDivisionError exception.
    """
    return x % y

if __name__ == '__main__':
    n: int = 13
    m: int
    m = 13.2
    print(f'{{__doc__=!s}}', sep="", end="\n*****\n")
    print(f'{{__annotations__=!s}}', sep="", end="\n*****\n")
    print(f'{{f.__doc__=!s}}', end="\n*****\n")
    print(f'{{f.__annotations__=!s}}', end="\n*****\n")
    print('help(f):')
    help(f)
    print("*****\nThe End")
```

буде виведено:

```
__doc__=
This module is designed to demonstrate module and function documenting.

*****
__annotations__={'n': <class 'int'>, 'm': <class 'int'>}
*****
f.__doc__=
    Return the remainder.

    Keyword arguments:
    x - dividend
    y - divider (default 2)

    pre: y!=0
        If not so, raise ZeroDivisionError exception.
```

```

****
f.__annotations__={'x': <class 'int'>, 'y': <class 'int'>, 'return':
<class 'int'>}}
****
help(f):
Help on function f in module __main__:

f(x: int, y: int = 2) -> int
    Return the remainder.

    Keyword arguments:
    x - dividend
    y - divider (default 2)

    pre: y!=0
        If not so, raise ZeroDivisionError exception.

****
The End

```

Додамо, що за анотацією **m: int** змінна **m** утворена не буде (і оголошенням змінної ця анотація також не є). Такий прийом використовують тоді, коли відомо, що змінна в програмі має бути, але початкове значення вона отримає згодом. Також у наведеному коді змінній **m**, анотованій як ціле, навмисно присвоєно дійсне значення. Це демонструє той факт, що відповідність анотацій коду залишається виключно на сумлінні програміста. Зрозуміло, що в реальних програмах так робити не слід.

◀

Примітка. Функція **help** під час виведення відкидає білі рядки на початку та в кінці. Крім того вона змінює абзацні відступи: за першим не білим фізичним рядком вона визначає абзацний відступ у рядку документації, під час виведення вона в кожному фізичному рядку замінює цей абзацний відступ на 4 пробіли.

6.12 Довгі функції

Багато авторів функцію, що складається більше ніж з 5-10 рядків²⁸ вважають *довгою*. Дехто знижує планку до 3-5. Але всі вважають **довгі функції** ознакою **поганого** коду. Чому ж таки довгі функції погані?

По-перше, якщо функція складається з великої кількості інструкцій, то стає громіздкою і людині з її кодом стає важко працювати (хоча б тому, що її тіло може не вміщуватися на один монітор). По-друге, велика кількість інструкцій передбачає складну логіку — і це теж не спрощує роботу людини з кодом. Якщо довгу функцію записати через менші допоміжні функції, кожна з яких виконує **одну** конкретну задачу, то такий код буде набагато зрозумілішим. Крім того, його окремі частини можна буде окремо змінювати, тестувати, налагоджувати — і це теж великий плюс. Насамкінець, довгий код найчастіше приховує в собі явне чи неявне дублювання, що теж не покращує якість коду.

Єдиний випадок, коли довгі функції мають право на життя, — це функції, що реалізують складні математичні алгоритми, які не можуть бути розкладені на підзадачі, або через їх логіку, або через критерії ефективності коду.

²⁸ У випадку мови Python означення варто з підрахунку цієї кількості виключати.

Висновок. Довгих функцій слід уникати. За їх появи треба проводити декомпозицію коду. Узагалі функції мають бути короткими і мати єдине призначення.

Примітка. Те саме стосується і модулів: якщо не враховувати означення, що в них містяться, то залишок також не повинен бути довгим.

6.13 Структура консольного застосування та функціональне проєктування

Приклад. Наступна програма обчислює значення квадратичної функції у заданій точці.²⁹

```
"""
This program calculates the value of given quadratic function at given
point.

All data are entered by user.
"""

def input_real(prompt=None):
    return float(input(prompt))

def input_quadratic():
    print("Input the coefficients of a*x^2+b*x+c as real. Each on its
own line.")
    a = input_real("a: ")
    b = input_real("b: ")
    c = input_real("c: ")
    return a, b, c

def calculate_quadratic(a, b, c, x):
    """
    Calculate the value of quadratic function a*x^2+b*x+c at point x.
    """
    return a * x * x + b * x + c

def quadratic2str(a, b, c):
    """
    Return string representing quadratic function a*x^2+b*x+c.
    """
    return f'{a}*x^2{b:+}*x{c}'

print("We are going to calculate quadratic function a*x^2+b*x+c at point
```

²⁹ У запропонованій Дональдом Кнотом та широко розповсюдженій у науковому світі нотації, яка використовується в редакторі TeX (LaTeX, тощо) запис x^y виглядає як x^y або $x^{\{y\}}$, x^{2y} записується як $x^{\{2y\}}$, тобто знак $^$ використовується для позначення верхнього індексу. Для позначення нижнього індексу використовується знак $_$ (підкреслення): x_y можна записати $x_{\{y\}}$ або $x_{\{y\}}$, x_{i+1} записується так $x_{\{i+1\}}$. Оскільки користувач програми не повинен знати, якою мовою вона написана, то в друкованих повідомленнях не варто прив'язуватися до позначення операцій саме в мові Python. У наведеному прикладі використовується TeX-нотація.

```
x.")  
  
a, b, c = input_quadratic()  
x = input_real("and now input real x: ")  
  
fx = calculate_quadratic(a, b, c, x)  
  
print(f'for {x=} {quadratic2str(a,b,c)} = {fx}')
```

Її структура достатньо стандартна для консольного застосування і поділяється на чотири логічні частини, що не перетинаються.

Спочатку виводяться відомості про саму програму. До них можна віднести автора, версію, загальне призначення програми. У наведеному прикладі на початку програма повідомляє своє призначення.

Далі виконується введення даних. У кінці введення для контролю інколи виводять увведені значення. У цій програмі це може здаватися надлишковим, але бувають ситуації (наприклад, зчитуються якісь складні дані або треба мати впевненість, що під час обробки зчитані вхідні дані випадково не були змінені), коли це має сенс.

Після введення виконуються обчислення. Зокрема, обчислюється значення, яке треба повідомити користувачу.

Наприкінці результат обчислень повідомляється користувачу. У проміжку між введенням та повідомленням результату могло відбуватися багато чого, тому разом з результатом доцільно нагадати користувачу й ті дані, для яких відбувалися обчислення.

Здавалося б, що виконати обчислення можна було б у тій самій інструкції, що виводить результат обчислення. Але це **дуже хибний** підхід, який не покращує код. Якщо код логічно розбито на частини *Введення даних*, *Обчислення*, *Виведення результатів*, то кожна з цих частин може не тільки окремо розроблятися, тестуватися та налагоджуватися, але зміни в одній не спричинятимуть змін в інших (особливо випадкових). Крім того, якщо буде необхідно змінити джерело надходження даних у програму або його формат, то це можна буде легко зробити незалежно від інших частин коду. Якщо буде необхідно якимось інакше виводити дані, наприклад, не на консоль, то це теж простіше зробити, коли три зазначені частини між собою не перетинаються. У наведеному прикладі, зазначені частини записані за допомогою допоміжних функцій.

Ознаки поганого коду можна ліквідовувати або можна запобігати їх утворенню. Одним з методів запобігання є функціональне проєктування, яке забезпечує кращий з точки зору модифікації, наладки, тестування та повторного використання код. Функціональне проєктування можна не тільки використовувати для первинної побудови коду, але й під час модифікацій.

Згідно з парадигмою **функціонального проєктування** (англ. *functional design*) кожна складова частина повинна мати рівно один **обов'язок** (те, безпосередньо за що вона відповідає; її призначення) та виконувати його з мінімальним впливом на інші частини. **Принцип єдиної відповідальності** (англ. *Single Responsibility Principle*) також підтримує те, що кожний логічний фрагмент коду (зокрема, функція) повинен мати єдину відповідальність.

Переваги функціонального проєктування полягають у тім, що фрагменти коду (функції, модулі, класи) з єдиним призначенням набагато простіші, а тому легші для сприйняття та менш витратні для проєктування, реалізації, модифікації, підтримки. За відсутності побічних ефектів (впливу на інші частини) кожна така частина може розроблятися, тестуватися та змінюватися окремо від інших. Зокрема, виправлення окремих помилок, наприклад, в обчисленнях, не буде тягнути за собою утворення нових помилок при виведенні обчислених даних, оскільки безпосереднє обчислення та безпосереднє виведення розташовано в різних фрагментах коду.

Крім того, у такому випадку легко змінити алгоритм обчислення, не порушивши при цьому виведення, та за потреби можна перенаправити виведення, не вносячи змін в обчислення.

Якщо деякий фрагмент коду виконує кілька дій одночасно, причому ці дії виконуються зовсім не в один виклик допоміжних функцій, то кажуть, що такий код *перевантажений обов'язками*. Перевантажені обов'язками функцій зазвичай є довгими. Спроба виконувати обчислення в інструкції, що задає виведення, є типовим прикладом перевантаженого обов'язками коду. Також доволі розповсюджена помилка проєктування — виводити значення з функції, що має їх тільки обчислювати.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: підпрограма, функція, модуль, пакет, виклик функції, інструкція виразу, аргумент виклику, позиційні аргументи, ключові аргументи, підказка на введення, заголовок функції, тіло функції, перелік формальних параметрів, фактичні параметри, проста інструкція, складена інструкція, інструкція **pass**, інструкція **return**, інструкція означення функції, блок (у програмуванні), локальна змінна, глобальна змінна, значення параметра функції за умовчанням, перевантаження імені, рядок документації, анотація, довга функція, функціональне проєктування, перевантажений обов'язками код, принцип єдиного обов'язку.

2. У чому переваги коду, написаного з використанням підпрограм, над кодом, що підпрограми не використовує?

3. Чому дублювання коду слід уникати?

4. Який спосіб уникнення дублювання коду описано в цьому розділі?

5. Яку семантику має інструкція виразу?

6. Чи може виклик функції одночасно містити позиційні та ключові аргументи?

7. Які аргументи мають записуватися першими в переліку аргументів виклику?

8. Чи важливий порядок ключових аргументів у виклику функції?

9. Чи можна для одного того самого формального параметра в переліку аргументів виклику двічі вказати значення (наприклад, один раз як позиційного і вдруге як ключового)?

10. Коли доцільно використовувати ключові аргументи?

11. Чи можна забезпечити, щоб деякий аргумент передавався у функцію тільки як ключовий?

12. Яку семантику має знак / у заголовку та виклику функції?

13. Яку семантику має знак * у заголовку та виклику функції?

14. Як забезпечити в програмі введення цілого значення?

15. Для чого використовують підказки на введення?

16. Якщо в підказці на введення просили ввести ціле число, то чи дає це гарантії того, що буде введено саме ціле число?

17. Як можна ввести дійсне значення $+\infty$?

18. Як можна ввести дійсне значення **NaN**?

19. Чому код консольної програми варто розділяти на три частини: введення даних, обчислення, виведення результатів?

20. Чим відрізняються бібліотеки **math** та **cmath**?

21. Який синтаксис має інструкція означення функції?

22. Де в програмі можна записувати означення функцій? Які умови мають завжди виконуватись?

23. Який синтаксис та семантику має інструкція **return**?

24. Чи продовжується виконання тіла функції за виконання інструкції **return**?

25. Яку синтаксичну будову мають складені інструкції?

26. Які дії задає інструкція **pass**?
27. Чи існують обмеження на типи аргументів виклику функції?
28. Чи існують обмеження на тип результату функції?
29. Чи може функція мати аргументів виклику менше, ніж кількість її формальних параметрів?
30. Яке значення повертається з функції, якщо воно не було явно задане програмістом?
31. Чи обов'язково виконання функції має завершуватися інструкцією **return**?
32. Чи можна для різних сутностей використовувати однакові імена? Якщо так, то за яких умов?
33. Чи може в тексті програми інструкція виклику функції **f** передувати інструкції означення функції **f**?
34. Опишіть синтаксис та семантику значень за умовчанням для параметрів функції.
35. Коли та скільки разів обчислюють значення за умовчанням для параметрів функції?
36. Опишіть як відбувається виклик функції.
37. Чи можливе у мові Python перевантаження імен? Що буде відбуватися за повторного означення імені?
38. Для чого програмісти документують код?
39. Що варто вказувати в документації до коду?
40. Які синтаксичні засоби можна використовувати для документування?
41. Що таке рядок документації? Яку інформацію в ньому зазначають?
42. Коли обчислюється рядок документації функції?
43. Як синтаксично виглядає анотація?
44. Для чого використовують анотації?
45. Чи перевіряє інтерпретатор відповідність коду його документації?
46. Чи доступна документація на функцію до її першого виклику?
47. Чому довгих функцій слід уникати?
48. Як можна позбутися від довгих функцій?
49. У чому переваги функціонального проектування?

Задачі та вправи

1. Що буде виведено за виконання програми?

```
def f(a):
    a += 1
    return a

def g(x):
    x = 2
    return x

a = 3
b = 44
print(f(a), g(b), a, b)
```

```
def f1(x, y):
    z = x + y
    x += 1
    y -= 1
    tmp = f2(z, x, y)
    print(x, y, z)
    return tmp

def f2(x, y, z):
    tmp = x + y + z
    x = y = z = 0
    return tmp

x = 0
y = 1
z = 2
```

```
def f1(x, y, z):
    x -= 1
    y += 1
    z += 2
    print(x, y, z)
    tmp = f2(x, y, z)
    print(x, y, z)
    return tmp

def f2(x, y, z):
    x, y, x = x, z, y
    print(x, y, z)
    return x + y + z

x = 0
y = 1
z = 2
```

	res = f1(y, x) print(x, y, z, res)	res = f1(x, y, z) print(x, y, z, res)
a)	б)	в)
2. Указати помилки, наявні в коді.		
def f1(a, b, c=3): pass f1(1, b=1, c=3) f1(1, c=1, c=3) f1(1, b=1) f1(1, 2, b=1) f1(b=1) f1(b=1, 1) f1(b=1, a=1) f1(c=5, b=1, a=1)	def f2(a, /, b, *, c=3, d=4): pass f2(4, d=5) f2(a=4, 3, d=5) f2(4, 3, d=5) f2(4, b=3, d=5) f2(1, 2) f2(1, 2, 4) f2(1, 2, d=4, c=5)	
a)	б)	

3. Написати програму, яка за довжинами сторін трикутника обчислює його периметр. Вхідні значення вважати коректними.

4. Написати програму, яка запитує в користувача координати точки дійсної площини, а потім виводить отриману точку. Зверніть увагу на зовнішній вигляд друкованого результату.

5. Написати вираз мови Python, відповідний математичному виразу

а) $\ln \sqrt{a^2 + b^2}$; б) $\sin((a + b)^{1/7})$; в) $\cos \sqrt[3]{a^{12} + b^{12}}$; г) $\operatorname{ctg}|\sqrt{a} - \sqrt{b}|$.

6. Написати вираз мови Python, відповідний математичному виразу $\log_x y$.

7. Написати програму, яка виводить

а) значення $\sqrt{2}$ у форматі з фіксованою крапкою з трьома знаками після крапки;

б) значення $(2,5)^{1,6}$ у нормалізованому форматі з чотирма знаками після крапки;

в) суму чисел π та e у форматі з фіксованою крапкою з двома знаками після крапки.

8. Попередня задача, але кількість знаків після крапки має задавати користувач.

9. Написати програму, в якій використано принаймні 7 різних функцій з модуля **math**.

10. Написати функцію, що обчислює значення виразу $e^x + 2yz$, та програму, що її використовує.

11. Написати функцію, що обчислює значення виразу $\frac{x+\pi}{x^2+y^4+e}$, та програму, що її використовує.

12. Розв'язати задачу 3 з використанням функції, що вводить координати точки, та функції, що виводить координати точки.

13. Написати

а) функцію, що вводить координати точки тривимірного простору;

б) функцію, що виводить координати точки тривимірного простору;

в) програму, яка запитує в користувача координати точки тривимірного простору, а потім виводить отриману точку.

Зверніть увагу на зовнішній вигляд друкованого результату.

14. Написати функцію, яка за дійсними коефіцієнтами a , b рівняння $ax + b = 0$ виводить рівняння на екран. Зверніть увагу на зовнішній вигляд друкованого рівняння.

15. Написати функцію, що за прізвищем та ім'ям друкує персональне вітання. Написати програму, яка запитує в користувача прізвище та ім'я, а потім друкує для нього персональне вітання.

16. Написати функцію, що за днем, місяцем та роком повертає рядок з відповідною датою у форматі **dd.mm.yy**. Функція має контролювати зміст своїх аргументів виклику.

17. Як у програмі можна задати дійсні значення **NaN**, $+\infty$, $-\infty$?

18. Написати функцію, що обчислює значення функції

$$f(a, x) = |x^5 + |ax - x^3| - a| + ax^2 + a^8$$

та програму, що її використовує.

19. Написати функцію, що за довжиною кола обчислює його радіус. Написати функцію, що за радіусом обчислює площу круга. За допомогою зазначених функцій написати програму, яка запитує в користувача довжину кола (додатне дійсне число), а у відповідь повідомляє радіус та площу обмеженого нею круга. Вважати, що користувач задає довжину кола коректно.

7 Керування порядком обчислень

7.1 Умови

7.1.1 Операції порівняння

Двомісні операції **порівняння** позначаються операторами `>`, `<`, `>=`, `<=`, `==`, `!=` («більше», «менше», «більше або дорівнює», «менше або дорівнює», «дорівнює», «не дорівнює»). Результатом порівняння є **False** («хибність») або **True** («істина») — значення логічного типу **bool**.

Порівняння цілих значень відбувається за законами математики. При цьому варто пам'ятати, що за перетворення логічних значень до типу цілих **False** стає **0**, а **True** — **1**. При зворотному перетворенні ненульові значення дають **True**, а значення **0** (нуль) — **False**.

Порівняння дійсних. Порівняно з математикою, де кожне значення має дорівнювати само собі, існує рівно одна відмінність³⁰: **NaN** $\neq$ **NaN**. Решта порівнянь за участю **NaN** дають цілком очікуваний результат: **False**.

Порівняння комплексних чисел. Аналогічно математиці, до комплексних значень застосовні тільки порівняння на рівність та нерівність (оператори `==`, `!=`).

Порівняння числових значень різних типів. Числові типи **bool**, **int**, **float** сумісні між собою в порівняннях. Мова Python дозволяє порівнювати дійсні значення з великими за абсолютною величиною цілими, що не можуть бути навіть наближено зображені в типі **float**.

Порівняння рядків відбувається лексикографічно, окремі символи впорядковуються згідно їх unicode-кодів. Так, `'abc' < 'AXcd'` має значення **False**, бо в кодовій таблиці малі латинські розташовуються після великих латинських, а `'987' < 'Aa'` має значення **True**, бо код символу **9** менший за код символу **A**. Рядки, що починаються з кириличної літери будуть менші за довільний рядок, що починається з літери з набору US-ASCII (серед них: десяткові цифри, підкреслення, знаки пунктуації, пробіл, табуляція, символи, латинські літери). До рядків застосовні всі оператори `>`, `<`, `>=`, `<=`, `==`, `!=`.

Порівняння значень довільних типів. У мові Python порівнювати на рівність та нерівність можна значення довільних типів. Якщо для деякого типу не задано, як виконувати порівняння, то перевірки `==` та `!=` будуть виконуватися за ідентифікаторами (identity) об'єктів. Спроба порівняти за допомогою `>`, `<`, `>=`, `<=` (якщо відповідні порівняння для типу не означені) призведе до виняткової ситуації (і можливого аварійного завершення програми).

Приклади

1. Вирази `1 == 2`, `'A' == 'a'` і `-1.4 > True` мають значення **False**, вирази `1 != 2`, `8 >= 8`, `13 > True` і `False < True` — значення **True**.

2. Значенням виразу `13 == '13'` є **False**, за спроби виконати в інтерпретаторі порівняння `13 <= '13'` буде виведено повідомлення про помилку. ◀

7.1.2 Перевірка властивостей

У мові Python крім порівнянь можна перевіряти³¹:

- чи є об'єкти ідентичними;
- чи є один об'єкт елементом іншого (інший об'єкт має бути контейнером);
- чи є один рядок підрядком іншого;
- чи є об'єкт екземпляром заданого класу.

³⁰ Насправді це регламентує стандарт IEEE 754.

³¹ А також чи є один клас похідним від іншого, але про це не зараз.

Перевірка ідентичності об'єктів може бути записана з використанням оператора **is**. Він повертає істину тоді і тільки тоді, коли його аргументи мають однакову «ідентичність» (identity), тобто є одним тим самим об'єктом.

Ідентифікатор³² (англ. *identity*) об'єкта повертає вбудована функція **id**. Результатом виразу **id(obj)** є ціле число, яке є «ідентичністю» значення **obj**. Вираз **x is y** задає ту саму перевірку, що й вираз **id(x) == id(y)**.

Так, якщо ім'я **var** означено, то значенням виразу **var is var** буде **True**. Значення виразів **1_000_000 is 1_000_000** та **15.5 is 15.5** точно може бути обчислено тільки інтерпретатором під час виконання: залежно від того, який тип значень та як вони утворювалися, однакові значення можуть зберігатися як у різних об'єктах, так і в одному тому самому об'єкті.

Приклад такої ситуації можна побачити при роботі з інтерпретатором.

x = 1000000
y = 1000000
x is y
False
x == y
True

Змінні **x** та **y** мають однакові значення, але зв'язані з різними об'єктами. ◀

Узагалі, при побудові власних типів варто притримуватися правила: **ідентичні об'єкти мають бути рівними**. Для бібліотечних типів воно виконується (за винятком значення **NaN**). Додамо, що інтерпретатор цю вимогу не контролює. Ще раз зазначимо, що обернене виконується не завжди: рівні об'єкти **можуть не бути** ідентичними.

Оператор **is not** позначає заперечення оператора **is**: для ідентичних об'єктів він має повертати **False**.

Незважаючи на те, що однакові значення можуть зберігатися в різних об'єктах, під час виконання існує не більше одного об'єкта, що зберігає значення **None** типу **NoneType**, отже значенням **None is not None** гарантовано буде **False**.

Перевірка належності здійснюється операторами **in** та **not in**. Для контейнерів (відповідні типи розглядатимуться пізніше) оператор **in** задає перевірку входження елемента в контейнер. Для рядків він має семантику «є підрядком». Так, значенням виразу **'123' in '012354'** є **True**, бо рядок **123** є підрядком рядка **012354**; значенням виразу **'321' in '012354'** є **False**.

Оператор **not in** реалізує протилежну перевірку: **'123' not in '012354'** має значення **False**, **'321' not in '012354'** має значення **True**.

Чи зберігає об'єкт екземпляр (=значення) заданого класу можна визначити за допомогою вбудованої функції **isinstance**: вираз **isinstance(object, one_class)** задає перевірку того, що значення **object** є екземпляром класу **one_class** або похідних від нього. Функція **isinstance** дозволяє виконати перевірку відносно декількох типів відразу: для цього можна використати вираз вигляду **isinstance(object, (перелік типів))**.

Приклади перевірки типів наведено нижче.

isinstance(False, bool)
True
isinstance(False, int)
True
isinstance(15.4, (int, float, complex))
True

Останню перевірку, починаючи з Python 3.10, можна записати так.

³² У CPython це буде адреса, за якою розміщено об'єкт.

```
isinstance(15.4, int|float|complex)
```

```
True
```

Тип **bool** є похідним від **int**, тому значенням виразу **isinstance(False, int)** є **True**. З практичної точки зору це означає те, що значення **False** можна використовувати всюди, де можна використовувати цілі числа.

Перевірка типу може бути корисна, коли залежно від типу аргументу функція має виконувати різні дії. Таке використовують тоді, коли функція має обробляти аргументи різних типів, але оформити кілька різних функцій синтаксично неможливо або небажано через засмічення інтерфейсу схожими функціями. ◀

7.1.3 Логічні операції

Двомісні операції «і», «або» та одномісна «не» (відповідно, булеве множення, або кон'юнкція, булеве додавання, або диз'юнкція, та заперечення) у логіці та математиці називаються *булевими*, або *логічними*, й застосовуються до булевих значень. Їх означення наведені в табл. 13.

Табл. 13 — Логічні операції

A	B	A і B	A або B	не A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

У мові Python їм відповідають оператори **and**, **or** та **not**, які, як і в більшості сучасних мов програмування, можуть застосовуватися не тільки до логічних значень. У такому випадку на основі правил приведення до логічного типу виконується оцінка істинності аргументів. Відмінністю від математики та багатьох інших мов програмування є те, що тільки **not** завжди повертає значення логічного типу.

Перш за все, під час оцінки істинності значення **False**, **None**, нулеві числові значення, порожні рядки та порожні контейнери стандартних типів завжди оцінюються як **False**; решта значень оцінюється як **True**.

Під час обчислення виразу **not expr** обчислюється значення *expr*, якщо воно оцінюється як **True**, то значенням виразу є **False**, інакше — **True**. В обох випадках обчислення оператора породжує логічне значення.

Значенням виразів **not False**, **not None**, **not 0**, **not 0.0** є **True**; значенням виразів **not True**, **not 15**, **not -1.5** є **False**.

Інтерпретатор Python виконує так зване *ледаче*³³, або *скорочене*, обчислення булевих операцій **and** та **or**. Не в усіх випадках будуть обчислені обидва операнди. Спочатку обчислюється їх перший операнд. Якщо для операції **and** він оцінюється як **False**, то другий операнд не обчислюється. Аналогічно, якщо перший операнд операції **or** оцінюється як **True**, то другий операнд не обчислюється. Результатом булевої операції стає останній обчислений операнд.

Приклади наведені в табл. 14.

Табл. 14 — Приклади обчислення виразів з логічними операціями у мові Python

³³ Принцип ледачих обчислень: «Обчислюю тільки те, що треба». Девізом же енергійних обчислень є: «Обчислюю все, що можу».

Вираз	Значення	Примітка
False and 13	False	
0 and (0/0)	0	0/0 обчислюватися не буде, отже помилка часу виконання не виникне
13 or 15.5	13	
15.5 or 13	15.5	
None or 15	15	
None or ""	""	
"" and 1	""	
'123' and 15.5	15.5	

Також зазначимо, що обчислення саме **and** та **or**, а не їх аргументів, нових об'єктів не породжує.

Приклад. Проведемо відповідний експеримент в інтерпретаторі:

```
x = 1500000
xx = 1500000
x is xx
```

```
False
```

```
y = 1700000
z = x and y
z is y
```

```
True
```

```
z = x or y
z is x
```

```
True
```

7.1.4 Пріоритети

Пріоритети операторів мови Python і в цій частині дещо відрізняються від прийнятих у мові C/C++ (і не тільки). Оновимо таблицю пріоритетів (див. табл. 15).

Табл. 15 — Пріоритети операторів мови Python

Оператор	Примітка
**	Має властивість правобічного зв'язування.
унарні +, -	
мультиплікативні *, /, //, %	Мають властивість лівобічного зв'язування
адитивні бінарні +, -	Мають властивість лівобічного зв'язування
in, not in, is, is not, <, <=, >, >=, !=, ==	Усі рівнопріоритетні. Можливе утворення ланцюжків.
not	
and	Виконуються ледачі обчислення зліва направо.
or	Виконуються ледачі обчислення зліва направо.

Зверніть увагу, що заперечення має доволі низький пріоритет і вираз **not expr1+expr2** буде обчислюватися як **not (expr1+expr2)**, а оператори **<, <=, >, >=, !=, ==** рівнопріоритетні. Крім того, синтаксис мови Python дозволяє утворення ланцюжків такого типу

expr1 op1 expr2 op2 ... opN exprN

де **expr1, expr2, ... exprN** довільні вирази, а **op1, op2, ... opN** — довільні оператори із групи **in, not in, is, is not, <, <=, >, >=, !=, ==** (не обов'язково однакові). Наведений вираз еквівалентний такому

(expr1 op1 expr2) and (expr2 op2 expr3) and ... and (exprN_1 opN exprN),

за тим винятком, що кожен з виразів **expr1, expr2, ... exprN** обчислюється не два рази, а один.

Приклади

1. Враховуючи пріоритети операторів, вираз **(k > 0) and (d > 0)** можна записати як **k > 0 and d > 0**, але перша форма запису сприймається людиною краще. Узагалі, намагайтеся записувати вирази так, щоб їх «зміст» не приховувався відсутніми дужками.

2. Математичну умову $x < y < z$ мовою Python можна записати як **x < y < z**.

3. Математичну умову $x \in (1; 3]$ можна записати мовою Python як **1 < x <= 3**.

4. Математичну умову $x=y=z$ того, що числа x, y, z попарно рівні між собою можна записати мовою Python як **x == y == z**.

◀

Записуючи вирази, не зловживайте пріоритетами операторів! Час, що економиться на дужках у виразах, потім витрачається на розшифрування логіки виразів та пошук помилок.

Зауваження. Замість виразів вигляду **not (expr1 in expr2)** та **not (expr1 is expr2)** PEP8 рекомендує використовувати **expr1 not in expr2** та **expr1 is not expr2** відповідно.

7.1.5 Умови

Вирази, які можуть оцінюватися як істинні, або хибні, називаються **умовними**, або **умовами**. У програмуванні вони відіграють особливу роль, оскільки є основою для прийняття рішень з вибору подальшого шляху обчислень. Значення умовного виразу не обов'язково має належати логічному типу, але для нього обов'язково має існувати механізм оцінювання його істинності (приведення до типу **bool**).

Наприклад, умовою того, що рівняння $ax + b = 0$ з коефіцієнтами a та b має єдиний розв'язок, є математична нерівність $a \neq 0$. Залежно від значення умови можна обчислити розв'язок рівняння або з'ясувати, чи має воно нескінченно багато розв'язків, чи не має жодного (для цього може знадобитися умова $b = 0$).

Умову часто використовують як **ознаку** деякої властивості. Ознака істинна, якщо властивість має місце, інакше хибна. Наприклад, умова $a \neq 0$ є ознакою того, що рівняння $ax + b = 0$ має єдиний розв'язок.

Найпростіші умови дуже часто мають вигляд порівнянь. Наприклад, умовою того, що значення цілої змінної **a** парне (ділиться на 2 без остачі), є те, що воно дає остачу 0 від ділення на 2: **a%2 == 0**. Ту саму умову можна записати і так **not (a%2)**. Також умови можуть містити виклики функцій, наприклад, вираз **isinstance(var, int | float)** задає умову того, що значення змінної **var** є цілим або дійсним числом.

Складніші умови формують як системи або сукупності, складені з простіших умов. **Системи умов** записують мовою Python за допомогою оператора **and**, **сукупності** — за допомогою **or**. Також може використовуватись оператор **not**.

Приклади

1. Умовою того, що число x належить проміжку $[a; b]$, є математичний вираз $a \leq x \leq b$. Виразимо її так: **a <= x <= b**.

2. Припустимо, що значення числових змінних **a, b, c** зображують довжини відрізків. Умовою того, що з відрізків можна утворити трикутник, є система нерівностей:

$$a \geq 0, b \geq 0, c \geq 0, a+b > c, a+c > b, b+c > a.$$

Цій системі відповідає такий вираз (дужки не обов'язкові, але додають наочності).

```
(a >= 0) and (b >= 0) and (c >= 0) \
and (a+b > c) and (a+c > b) and (b+c > a)
```

Ця умова, хоч і відповідає математичній, але є прикладом **довгої** умови, що **погано**. Довгі умови краще ділити на логічні частини, які оформлюються як функції, і далі складати умову з викликів функцій.

У розв'язуваному прикладі перші три нерівності задають перевірку того, що значення змінних **a**, **b**, **c** можна розглядати як довжини відрізків. Решта нерівностей задає перевірку нерівності трикутника.

Перевірку нерівності трикутника, тобто частину **(a+b > c) and (a+c > b) and (b+c > a)**, без жодних вагань можна виділити в окрему функцію **triangle_inequality**.

```
def triangle_inequality(a, b, c):
    return (a+b > c) and (a+c > b) and (b+c > a)
```

Тоді умова може бути переписана так:

```
(a >= 0) and (b >= 0) and (c >= 0) and triangle_inequality(a, b, c)
```

Виділяти перевірку додатності всіх трьох значень одночасно в окрему функцію тут було б помилкою: якщо ця функція буде отримувати три аргументи, то вона буде виконувати кілька незалежних задач одночасно, що не вважається гарним стилем. Якщо виходити зі змісту, то можна ввести функцію **is_length**, що перевіряє умову того, що значення може задавати довжину.

```
def is_length(x):
    return x >= 0
```

Тоді остаточно отримуємо умову.

```
is_length(a) and is_length(b) and is_length(c) and \
triangle_inequality(a, b, c)
```

Останній вираз для людини більш зрозумілий, бо виражає суть здійснюваних перевірок.

3. Той факт, що цілі ненульові числа **a**, **b**, **c** утворюють геометричну прогресію, математично можна виразити як $\frac{b}{a} = \frac{c}{b}$. Результатом цілочислового ділення є ціле, тому вираз **(b//a) == (c//b)** взагалі **не відповідає** математичній умові. Вираз **b/a == c/b** вирішує проблему з діленням цілих, але теж **не є прийнятним**. Справа в тому, що тип **float** може містити лише наближення справжніх числових значень $\frac{b}{a}$ та $\frac{c}{b}$, тому результат порівняння значень типу **float** може відрізнитися від математичного порівняння. (Більш того, у мові Python можлива ситуація, коли частка двох цілих буде більшою за найбільше значення типу **float**.) Щоб уникнути цього, запишемо вираз у математично еквівалентній формі: **bb = ac**. Отже, умова того, що числа **a**, **b**, **c** утворюють геометричну прогресію може бути записана так **b*b == a*c**.

4. Умову того, що рядок **s** порожній або містить KNU як підрядок можна записати так: **(not s) or ('KNU' in s)**. Функція **len** для рядка повертає його довжину. Тоді умова **3 <= len(s) <= 5** задає перевірку того, що довжина рядка **s** знаходиться в межах від 3 до 5 включно. Використання умов вигляду **len(s) == 0, len(s) != 0, s == "", s != ""** є нерекомендованим.

4

- Перш ніж записувати математичну умову мовою програмування, спробуйте провести математичні перетворення.

7.1.6 Використання перевірок **==**, **is** та **isinstance**

Мова Python має три перевірки, які в деяких випадках здаються доволі схожими. Це перевірка об'єктів на рівність, на ідентичність та перевірка типу об'єкта. Спробуємо підвести підсумки та розібратися, коли яку краще використовувати.

Перевірка на рівність (оператор `==`), якщо означена для типу, виконує порівняння значень об'єктів. Якщо не означена явно, то порівнює ідентифікатори об'єктів. Якщо важливо порівнювати саме значення, то слід використовувати її.

Перевірка на ідентичність (однаковості об'єктів) (оператор `is`) завжди порівнює ідентифікатори об'єктів. Вона має використовуватися тоді, коли треба перевірити, що значення зберігаються в одному тому самому об'єкті. Крім того вона виконується швидше за перевірку рівності, і її використання доцільно, якщо гарантовано, що одне те саме значення не може зображуватися в пам'яті різними об'єктами. Остання ситуація має місце для деяких вбудованих констант (наприклад, `None`), а також для класів. Отже, замість `int == float` краще писати `int is float`. Щодо порівняння з `None`, то для класів, що зображують структури даних, порівняння `==` може не відрізняти порожню структуру від `None` (в обох випадках елементів нема), тому згідно PEP8 порівняння з `None` завжди має записуватися через `is`, наприклад, `expr is None`.

Щодо явних порівнянь значень з `True` та `False`, то їх взагалі не слід писати (і PEP8 цілком це підтримує). Через особливість обчислення операторів `and` та `or` у мові Python вирази `expr == True`, `expr is True` та `expr` не є еквівалентними, і умові, зазвичай, відповідає саме остання форма, тобто `expr`. Відмінності між зазначеними виразами демонструє табл. 16.

Табл. 16 — Обчислення деяких умов у мові Python

Умова	Істиннісне значення	Примітка
<code>12 == True</code>	False	Значенням True є ціле 1, далі виконується порівняння цілих.
<code>1 == True</code>	True	
<code>1.0 == True</code>	True	Значенням True є ціле 1, далі виконується порівняння дійсного 1.0 та цілого 1 за звичайними математичними правилами.
<code>(2 or 13) is True</code>	False	Значенням (2 or 13) є об'єкт, що зображує 2 .
<code>1 is True</code>	False	Хоч тип bool і є похідним від int , а значення True відповідає цілій 1 , але об'єкти все-таки різні.
<code>12</code>	True	
<code>1</code>	True	
<code>2 or 13</code>	True	

Перевірка типу об'єкта може бути записана через **type** та оператор **is**, але цього PEP8 рекомендує не робити і використовувати виключно функцію **isinstance**.

7.2 Інструкція розгалуження

Рівняння $ax + b = 0$, залежно від конкретних значень a , b , може розв'язуватися одним з трьох шляхів. Шлях обирається після визначення, чи справджується умова $a \neq 0$; якщо це не так, то чи дійсна умова $b = 0$. Розглянемо засоби, які дозволяють указати вибір шляху обчислень залежно від тих або інших умов, і скористаємося ними в розв'язанні рівняння.

Вибір одного з двох можливих шляхів обчислення можна задати за допомогою *інструкції розгалуження* (умовної інструкції). Інструкція розгалуження є складеною інструкцією і в найбільш загальному вигляді має вигляд:

```
if умова_1 : набір_інструкцій_1
elif умова_2 : набір_інструкцій_2
...
elif умова_N : набір_інструкцій_N
else : набір_інструкцій_(N+1)
```

Слова **if**, **elif**, **else** є ключовими словами. Обов'язковою частиною є тільки секція **if**. Секції **elif** та **else** обов'язковими не є, кількість секцій **elif** може бути довільною.

Інструкція розгалуження виконується так. Послідовно обчислюються значення виразів **умова_1**, **умова_2**, ..., **умова_N** та відразу оцінюється їх істинність. Як тільки буде знайдено перший вираз, який є істинним, скажімо, **умова_k**, то виконується **набір_інструкцій_k** і виконання інструкції розгалуження завершується: решта умов не обчислюється, набори інструкцій з інших секцій не виконуються. Якщо жоден з виразів не є істинним, то виконується набір інструкцій **набір_інструкцій_(N+1)** з секції **else** (якщо наявна) і на цьому виконання інструкції також завершується.

Приклад. Розглянемо такі інструкції.

```
n = int(input('Enter integer: '))
if n%2 == 0:
    print('even')
else:
    print('odd')
```

Якщо введено парне значення **n**, то обчислення виразу **n%2 == 0** (перевірка умови) дає істинне значення, і виконується інструкція **print('even')**. За непарного значення жодна умова (у цій інструкції вона єдина) не виконується, і виконується інструкція **print('odd')**. ◀

Приклад. Написати фрагмент коду, який за дійсним x обчислює значення $f(x)$ та присвоює його змінній y .

$$f(x) = \begin{cases} x, & \text{якщо } 3 \leq x \leq 7, \\ x + 2, & \text{якщо } x > 7, \\ 2x, & \text{якщо } -5 < x < 3, \\ 0, & \text{якщо } x \leq -5. \end{cases}$$

Проведемо попередній аналіз та перепишемо формулу обчислення $f(x)$ в еквівалентному вигляді.

$$f(x) = \begin{cases} 0, & \text{якщо } x \leq -5, \\ 2x, & \text{якщо } -5 < x < 3, \\ x, & \text{якщо } 3 \leq x \leq 7, \\ x + 2, & \text{якщо } 7 < x. \end{cases}$$

Безпосередньо за формулою запишемо такий фрагмент коду.

```
if x <= -5: y = 0
if -5 < x < 3: y = 2*x
if 3 <= x <= 7: y = x
if 7 < x: y = x+2
```

Цей код є правильним, але **неоптимальним** за кількістю виконуваних операцій. Якщо значенням змінної x є **-9.0**, то обчислюються всі чотири умови, хоча з математичної точки зору зрозуміло, що за істинності умови $x \leq -5$ решта умов хибні. Отже, модифікуємо фрагмент коду.

```
if x <= -5: y = 0
elif -5 < x < 3: y = 2*x
elif 3 <= x <= 7: y = x
elif 7 < x: y = x+2
```

Тепер за значення **-9** обчислюється тільки перша умова. Нехай значенням змінної x є **0.1**. Тоді вираз $x \leq -5$ є хибним, і обчислюється вираз $-5 < x < 3$. Але $x \leq -5$ є хибним, тому $-5 < x$ є істинним! Отже значенням виразу $-5 < x < 3$ є значення $x < 3$. Міркуючи так само далі, отримуємо ще один варіант.

```
if x <= -5: y = 0
elif x < 3: y = 2*x # якщо ми тут, то -5 < x є істинним
elif x <= 7: y = x # якщо ми тут, то 3 <= x є істинним
else: y = x+2 # якщо ми тут, то 7 < x є істинним
```

Коментарі тут для реальних проєктів, звісно, надлишкові.

Зауважимо: такий фрагмент коду для нашої задачі є **помилковим**.

```
if x <= -5: y = 0
if x < 3: y = 2*x
if x <= 7: y = x
else: y = x+2
```

Якщо значенням x є 1.0 , то умова $x < 3$ істинна, і спочатку виконується присвоєння $y = 2*x$. Проте потім перевіряється умова $x \leq 7$, виявляється істинною, і виконується $y = x$, що, вочевидь, є помилковим. ◀

Приклад. Написати фрагмент коду, який за значенням b виводить `TRUE` за істинного значення, та `FALSE` за хибного.

```
if b:
    print('TRUE')
else:
    print('FALSE')
```

Зазначимо, що використання виразу $b == \text{True}$ в якості умови для цієї задачі є помилковим. ◀

Приклад. Наступна функція задає обчислення виразу $(-1)^n$ за цілим n .

```
def f(n):
    if n%2:
        return -1
    else:
        return 1
```

Тут використано той факт, що ненульові числові значення в умовах оцінюються як `True`, і замість умови $n\%2 == 1$ записано умову $n\%2$. Щодо дійсних аргументів, то функція може бути викликана з дійсним аргументом, але за наведеною спрощеною формулою обчислення проводити не можна, бо у загальному випадку значенням має бути комплексне число. ◀

Перевірку на те, чи є значення змінної x числового типу ненульовим можна виразити умовою x . Умова `not x` задає перевірку на те, що значення є нулем. Такого вигляду умови дуже популярні в програмах мовою C (навіть більше, явна перевірка на нуль в них виглядає дивакуватим). Але така перевірка вимагає виконання неявного перетворення типів. Чи записувати порівняння з нулем явно — справа уподобань: є аргументи на користь обох сторін. Якщо вважати, що явне краще за неявне, то тільки явне порівняння з нулем. Але якщо згадувати, що явне порівняння з нулем може вимагати неявного перетворення типів (наприклад, цілого до дійсного), то від явного порівняння можна відмовитися. Також, різні мови програмування мають різні стилі уподобання.

Щодо рекомендації PEP8, то про порівняння чисел з нулем там нічого не сказано, але для перевірки чи є рядок порожнім рекомендується використовувати не перевірку його на рівність з порожнім рядком, і не перевірку його довжини, а неявне приведення до типу `bool` (порожній рядок оцінюється як `False`).

7.3 Операція розгалуження

У мові Python наявна *умовна операція*, або *операція розгалуження*, яка записується так.

expr1 if condition else expr2

Виконується вона так. Обчислюється вираз **condition** і оцінюється його істинність. Якщо він істинний, обчислюється значення виразу **expr1** і воно стає результатом обчислення умовної операції. Якщо він хибний, то обчислюється значення виразу **expr2**, яке стає результатом у цьому випадку.

Умовний оператор **if-else** є тернарним, бо має три аргументи. Його пріоритет нижчий за пріоритет оператора **or**.

Приклади

1. Значенням виразу x **if $x \geq 0$ else $-x$** є модуль числового значення змінної x .

2. Значенням виразу **a if a<=b else b** є найменше зі значень числових змінних **a** та **b**. ◀

7.4 Кілька слів про стиль коду та значення, що повертаються з функцій

Раніше вже зазначалося, що виконувати порівняння (**==**, **!=**, **is**, **is not**) умовних виразів з логічними константами **True** або **False** є помилковим.

Якщо умовний вираз оформити у вигляді функції, то жодного сенсу повертати з функції значення саме типу **bool** нема. По-перше, у деяких випадках це буде вимагати явного перетворення типу. По-друге, явно порівнюватися з **True** або **False** воно все одно не повинно.

Нехай **x**, **y** позначають деякі змінні, а **expr** позначає деякий вираз. У таблиці 17 наведені типові стилістичні помилки (праворуч) і їх стилістично кращі еквіваленти (ліворуч), що типові для функцій, що обчислюють умови.

Табл. 17 — Стилiстичнi помилки в запису функцiй, що обчислюють умови, та їх уникнення

Краще писати так	Так писати НЕ треба	
<code>return x < y</code>	<code>if x < y: return True else: return False</code>	<code>if x < y: return True return False</code>
<code>return y <= x</code>	<code>if y <= x: return True else: return False</code>	<code>if x <= y: return True return False</code>
<code>return expr</code>	<code>if expr: return True else: return False</code>	<code>if expr: return True return False</code>
<code>return not(expr)</code>	<code>if not(expr): return True else: return False</code>	<code>if not(expr): return True return False</code>
<code>return not(expr)</code>	<code>if expr: return False else: return True</code>	<code>if expr: return False return True</code>

Повернення з функцiй цiлих **1** та **0** замість логічних значень **True** та **False** також не є гарним. Також треба пильнувати за тим, щоб

- якщо хоча б одна гілка обчислення функції явно повертає значення в інструкції **return**, то кожна гілка обчислення має завершуватися інструкцією **return**, що явно повертає значення.

Приклади функцiй з кількома гілками обчислень (і типовими стилістичними проблемами таких функцiй) наведені в таблиці 18.

Табл. 18 — Стилiстичнi нюанси функцiй, в яких є розгалуження обчислень функцiй

Код	Примітка
<code>def f(x): if x == 3: return None else: return 1/(x-3)</code>	ОК. Обидва шляхи виконання завершуються інструкцією return , що явно повертає значення. Єдиний недолік це те, що за більшої кількості умов вкладені «сходинки» почнуть залазити в монітор сусіда праворуч ☺.
<code>def f(x): if x == 3: return None return 1/(x-3)</code>	ОК. Спочатку розглядаємо специфічні випадки і завершуємо для них виконання, потім загальний випадок. Усі шляхи виконання завершуються інструкцією return , що явно повертає значення. Велика вкладеність сходинок виникати не повинна. Мабуть, <i>найкращий варіант</i> для функції, що виконує перевірку аргументів.

Код	Примітка
<pre>def f(x): if x != 3: return 1/(x-3) else: return None</pre>	Краще уникати. Зазвичай, загальний випадок (коли всі аргументи такі, як треба) розглядають останнім.
<pre>def f(x): if x == 3: return return 1/(x-3)</pre>	Стилістично погано.. Не всі інструкції return явно повертають значення, а тільки деякі.
<pre>def f(x): if x != 3: return 1/(x-3)</pre>	Стилістично погано. Не всі шляхи виконання завершуються інструкцією return , а тільки деякі.

Також варто уникати ситуацій, коли різні гілки обчислення функції за аргументів одних тих самих типів повертають значення різних типів (див. табл. 19). Наприклад, для дійсного аргументу одна гілка повертає дійсне значення, а інша — рядок. Проте, наведене у прикладах вище повернення значення **None**, як ознаки неможливості обчислення для певних значень аргументу, у багатьох випадках є цілком обміркованим і доречним.

Табл. 19 — Стилістичні нюанси функцій, в яких є розгалуження обчислень функцій

Код	Примітка
<pre>def f(x): if x == 3: return 'Ups' else: return 1/(x-3)</pre>	Зовсім стилістично погано. Бо за дійсного аргументу, залежно від його значення, функція може повернути і дійсне, і рядок.
<pre>def f(x): if x == 3: return 'Ups' return 1/(x-3)</pre>	Зовсім стилістично погано. По суті це майже повний аналог попередньої функції.
<pre>def f(x): if x == 3: print('Ups') else: return 1/(x-3)</pre>	Гірше придумати вже важко. Бо функція перевантажена обов'язками: обчислює та спілкується із зовнішнім світом. Крім того, а чи десь сказано, що вона буде використовуватись тільки в консольних застосуваннях та англійською мовою?

7.5 Використання інструкції **match** як інструкції вибору варіантів

Інструкція вибору варіантів наявна в багатьох мовах програмування. Схожа на неї інструкція **match** була запроваджена в Python 3.10. Проте інструкція **match** мови Python має більші можливості.

Суть інструкції вибору варіантів полягає в тім, що обчислюється значення виразу та послідовно порівнюється із заздалегідь заданими значеннями (варіантами), яким зіставлено деякий код; якщо значення співпало з одним із заданих, то виконується код, що відповідає цьому значенню. Різні мови програмування мають певні нюанси щодо того, як визначається код, що його слід виконати, але на інших мовах програмування зараз зосереджуватись не будемо. Натомість розглянемо як інструкція вибору варіантів може бути реалізована за допомогою інструкції **match** мови Python.

Інструкція **match** є складеною інструкцією, що складається з заголовку, за яким з абзацним відступом розташовується непорожня послідовність **case**-секцій, що задають можливі гілки обчислень.

Заголовок має вигляд

match вираз:

Кожна **case**-секція може мати один з двох виглядів

case вірець, з яким виконується зіставлення : набір інструкцій

або

case вірець, з яким виконується зіставлення if умова: набір інструкцій

Виконується ця інструкція так: обчислюється значення виразу, далі воно послідовно зіставляється зі вірцями. Як тільки буде знайдено вірець, з яким отримане значення може бути зіставлено і для цього вірця умова або відсутня, або істинна, то виконується відповідний набір інструкцій, після чого виконання інструкції **match** завершується (зокрема, зіставлення з подальшими вірцями не відбувається).

Вірці, з якими виконується зіставлення, у мові Python мають більше можливостей, ніж, скажімо, у мові C++. Але все-таки обмеження є. Зараз розглянемо тільки випадок, коли вірці задають значення і не спираються на внутрішню структуру отриманого значення.

В якості вірців можуть використовуватись літерали типів **int**, **float**, **str** (але не f-рядки), значення **True**, **False**, **None**. Також дозволяються вирази, що отримуються з числових літералів³⁴ застосуванням унарного мінусу. Якісь більш складні вирази або значення, задані звичайними³⁵ змінними, не дозволяються. Для вірців **True**, **False**, **None** використовується оператор **is**. Для решти зазначених вірців зіставлення виконується на основі порівняння **==**: якщо обчислене значення дорівнює значенню вірця, то вважається, що зіставлення відбулось.

Приклад вірців, що можуть використовуватись в інструкції **match**. (Звісно, що перелік не вичерпний.)

```
match m:
    case 4: pass
    case -4: pass
    case 5.6: pass
    case -5.6: pass
    case 'qwerty': pass
    case True: pass
    case False: pass
    case None: pass
```

Приклад. За виконання

```
x = 1
match x:
    case True: print(1)
    case 1.0: print(2)
    case 1: print(3)
```

буде виведено.

2

Третя **case**-секція не буде зіставлена за жодного значення змінної **x**, бо **1.0 == 1** є істинним, а тому є надлишковою. Приберемо її.

За виконання

³⁴ Комплексні літерали і вирази з ними та унарним мінусом теж можуть бути вірцями.

³⁵ Тобто такими змінними, що виникали на цей момент.

```
x = True
match x:
    case True: print(1)
    case 1.0: print(2)
```

буде виведено.

1

За виконання

```
x = True
match x:
    case 1: print(1)
    case True: print(2)
```

буде виведено.

1

Друга **case**-секція також не буде зіставлена за жодного значення змінної **x**, бо **True==1** є істинним.

Висновок з останнього прикладу: секції з вірцями **True**, **False** мають передувати секціям з числовими вірцями **1** та **0**.

◀

Приклад. Написати функцію, що за місяцем повертає пору року.

Використаємо інструкцію **match**.

```
def season(month):
    match month:
        case 1:
            res = 'winter'
        case 2:
            res = 'winter'
        case 3:
            res = 'spring'
        case 4:
            res = 'spring'
        case 5:
            res = 'spring'
        case 6:
            res = 'summer'
        case 7:
            res = 'summer' # (1)
        case 8:
            res = 'summer' # (2)
        case 9:
            res = 'autumn'
        case 10:
            res = 'autumn'
        case 11:
            res = 'autumn'
        case 12:
            res = 'winter'
```

return res # (3)

З виклику **season(7)** буде повернуто **'summer'**, бо виконається зіставлення з вірцем з рядка (1), далі виконається відповідний вірцю набір інструкцій (у прикладі це рядок (2)), після чого виконання інструкції **match** завершиться та буде виконуватись наступна за **match** інструкція, тобто інструкція з рядка (3).

Природньо задатися питанням: якщо один той самий набір інструкції відповідає кільком взірцям, то чи не можна уникнути дублювання? Для нашого випадку відповідь позитивна: взірці можна комбінувати з використанням знаку |.

Взірець можна записати у вигляді *взірець_1* | *взірець_2* | ... | *взірець_N*. У такому випадку під час зіставлення з ним будуть послідовно переглядатися *взірець_1*, *взірець_2*, ..., *взірець_N* поки або не буде знайдено зіставлення (і тоді вважатиметься, що зіставлення відбулося), або перелік взірців завершиться (і тоді зіставлення зі взірцем не відбулося). Модифікуємо код прикладу.

Приклад. Написати функцію, що за місяцем повертає пору року.

```
def season(month):
    match month:
        case 1 | 2 | 12:
            res = 'winter'
        case 3 | 4 | 5:
            res = 'spring'
        case 6 | 7 | 8:
            res = 'summer'
        case 9 | 10 | 11:
            res = 'autumn'
    return res
```

Наведена функція має недолік: вона не передбачає, що значення параметра **month** може не відповідати жодному взірцю. Зазвичай, так не роблять і в кінець інструкцію додають **case**-секцію, що здатна "перехопити" усі невраховані значення. Ця секція обов'язково має бути останньою. Ще раз модифікуємо приклад.

Приклад. Написати функцію, що за місяцем повертає пору року.

```
def season(month):
    match month:
        case 1 | 2 | 12:
            res = 'winter'
        case 3 | 4 | 5:
            res = 'spring'
        case 6 | 7 | 8:
            res = 'summer'
        case 9 | 10 | 11:
            res = 'autumn'
        case _:
            res = 'incorrect month'
    return res
```

Тепер за некоректного значення місяця буде повертатись рядок з відповідним повідомленням.

- Взірець `_` відповідає довільному значенню.

case-секції з умовами відносяться до специфіки Python. Використання умов може бути доцільним, коли є важливим тип отриманого значення або розглядається внутрішня будова значення. Але зараз обмежимося випадками, що відповідають класичній інструкції вибору варіантів.

Типова ситуація, в якій інструкція **match** може бути дуже корисна, — програмування різноманітних меню, обробників подій тощо. Жартівливий приклад з фрагментом коду, що використовує вибір варіантів, наведено нижче.

Згадаємо богатира, який стоїть на перехресті перед камінцем, на якому викарбовано: «Ліворуч підеш — коня втратиш, праворуч підеш — життя втратиш, прямо підеш — себе позабудеш.» Доступне богатиру «меню» можна задати за допомогою інструкції **match** так.

```
match direction:
    case 'ліворуч':
        втратити('кінь')
    case 'праворуч':
        втратити('життя')
    case 'прямо':
        позабути('себе')
```

7.6 Кілька слів про тестування

Коли за поставленою задачею було побудовано програму, виникає природне питання – чи правильно побудовано програму? Можливо, вона виконує зовсім не те, що треба. Кращого способу переконатися, ніж перевірити, не існує.

У реальних проєктах замовник формулює власні вимоги до програмного забезпечення, зазначаючи, що йому хотілося б отримати. Далі команда фахівців уточнює вимоги, будує технічне завдання та специфікацію майбутнього програмного продукту. У контексті цього можна перевіряти:

1. Чи відповідає програмне забезпечення вимогам замовника, тобто робить правильні та корисні для нього дії; чи повністю виконує те, що очікує від нього замовник (*you built the right product*). Таку перевірку називають *перевіркою правильності*, або *валідацією* (*validation*).

2. Чи правильно працює програмне забезпечення, тобто чи відповідає вимогам та специфікації проєкту (*you built the product right*). Цю перевірку називають *верифікацією*.

Для перевірки можна використовувати *тестування* та *формальну верифікацію* (формальне доведення відповідності коду специфікації). Зупинимось тільки на тестуванні.

Під тестуванням розуміють процес дослідження та випробування програмного продукту з метою оцінки якості, зокрема його перевірки та пошуку «вузьких» місць та небажаних ефектів (або переконання, що вони відсутні). Тестувати можна як окремі частини коду, так і їх взаємодію або систему в цілому.

Однією з технік тестування є виконання тестів, тобто виконання коду на заздалегідь підібраних наборах даних у заздалегідь визначених умовах за заздалегідь визначеним сценарієм. У найпростішому випадку тестовий приклад складається з вхідних даних та вихідних даних, які мають бути на них отримані; до опису тесту також включають опис функціональності, яка тестується, та необхідні дії для забезпечення можливості проведення тесту.

Для тестування простої програми на зразок задачі про обчислення значення функції як «чорної скриньки» (код та внутрішня будова програми вважається невідомою) слід скласти тестові приклади, що перевіряють:

- 1) правильність роботи програми за вхідних даних з області визначення виразу;
- 2) правильність роботи програми за вхідних даних не з області визначення виразу;
- 3) правильність роботи програми на вхідних даних, близьких до межі області визначення виразу.

Для тестування простої програми на зразок задачі про обчислення значення функції як «білої скриньки» (код та внутрішня будова програми вважаються відомими) слід побудувати набір тестів, що гарантує виконання кожної інструкції коду, тобто за виконання всіх тестів буде пройдена кожна наявна в програмі інструкція. Але це не є вичерпним тестуванням білої скриньки. З цим пов'язане поняття *покриття коду*, тобто визначення ступеня протестованості

вихідного коду програми. Можна вимірювати чи була виконана та протестована: кожна інструкція, кожна умова, кожен шлях виконання, кожна функція тощо.

У житті доволі часто виявляється, що множина даних для тестів «чорної скриньки» суттєво перетинається з множиною даних для тестів «білої скриньки».

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: оператори порівняння, оператор **is**, оператор **is not**, оператор **in**, оператор **not in**, оператор **not**, оператор **and**, оператор **or**, функція **id**, функція **isinstance**, функція **len**, логічні операції, ледачі обчислення, енергійні обчислення, умова, умовний вираз, інструкція розгалуження, умовна інструкція, інструкція **match**, умовна операція, операція розгалуження, валідація, верифікація, тестування, тестування як «білої скриньки», тестування як «чорної скриньки», тестовий приклад, покриття коду.

2. Які оператори порівняння наявні в мові Python? Значення якого типу вони породжують?
3. Як відбувається порівняння цілих чисел у мові Python?
4. Як відбувається порівняння дійсних чисел у мові Python?
5. Які порівняння застосовні до комплексних чисел?
6. Як відбувається порівняння рядків?
7. Які оператори порівняння застосовні до значень довільних типів? Як відбувається порівняння?
8. Які властивості можна перевіряти в мові Python?
9. Які перевірки задають оператори **is** та **is not**?
10. Які перевірки задають оператори **in** та **not in**?
11. Які оператори дозволяють перевірити входження одного рядка в інший?
12. У чому відмінність операторів **==** та **is**?
13. Чи може функція **isinstance** перевіряти належність екземпляра відразу декільком класам?
14. Як обчислюється оператор **not**? Значення якого типу він повертає?
15. Як обчислюється оператор **and**?
16. Як обчислюється оператор **or**?
17. Чи є оператор **and** мови Python комутативним?
18. Чи є оператор **or** мови Python комутативним?
19. Обчислення яких логічних операторів завжди призводить до утворення нового значення?
20. Як визначається тип значення, що є результатом обчислення логічних операторів?
21. Чи обов'язково умова повинна містити порівняння, перевірки властивостей чи логічні оператори?
22. Як можна покращити код, коли умова стає складною та/або малозрозумілою?
23. Чи можна використовувати явні порівняння на рівність та ідентичність з **True** та **False**? Чому?
24. Як краще записувати в коді перевірку типу об'єкта?
25. Вкажіть синтаксис умовної інструкції та порядок її виконання.
26. Які тернарні операції підтримуються мовою Python?
27. Як обчислюється умовна операція?
28. Значення якого типу може повертати функція, що обчислює умову?
29. Чи може функція за різних гілок обчислень повертати значення різних типів?
30. В яких випадках неявне повернення функцією значення **None** вважається поганим стилем?
31. Чим валідація відрізняється від верифікації?
32. Чим тестування «чорної скриньки» відрізняється від тестування «білої скриньки»?

33. Які тести слід обов'язково виконати за тестування програми, що обчислює значення функції?

Задачі та вправи

1. У чому відмінність виразів **a = 3**, **a == 3** та **a is 3**?
2. Обчислити значення виразів та вказати тип результату.
а) **1.5 < 7**; б) **'3' < 'a'**; в) **'Z' < 'a'**; г) **'77' < '119'**; д) **False < True**; е) **False < -1**?
3. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **not True == 0**; б) **not False or False == 1**;
в) **(2 > 1) and not True**; г) **not (6 < 3) and (7 < 5) or (3 == 5)**;
д) **True == 12**; е) **False < True** ж) **not 0 and 2**
4. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **'123' or 1 < 3**; б) **1 < 3 or '123'**;
в) **"" and (2 > 1)**; г) **15.5 and (61 < 13)**.
5. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **15.5 and (17 or 1 < 3)**; б) **15.5 and (17-17 or 2 < 3.5)**;
в) **5 or 7 < 14**; г) **1 < 2 < 3 and (6 < 13)**.
6. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **15.5 == 17 - 2.2**; б) **'5' == "5"**;
в) **5 == "5"**; г) **'1' < '2' < '3' == True**.
7. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **(2 < 3) == True**; б) **'5' in "5"**;
в) **5 is "5"**; г) **"1" < "2" < "3" == True**.
8. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **'2' < 3**; б) **'2' != 5**;
в) **5.5 == True**; г) **'3' == True**.
9. Обчислити значення виразів та вказати тип результату (якщо вираз є коректним).
а) **'on' not in 'not'**; б) **'th' in 'Python'**;
в) **isinstance(5, int)**; г) **isinstance(1.5, int)**.
10. Записати умову того, що змінні **x**, **y**, **z** позначають один той самий об'єкт.
11. Записати умову того, що значення змінних **x**, **y**, **z** попарно різні.
12. Записати умову того, що серед чотирьох значень:
а) усі попарно рівні; б) усі попарно різні; в) є принаймні два різних.
13. Записати умову того, що рядок складається рівно з однієї літери.
14. Записати умову того, що рядок складається рівно з однієї літери, яка
а) є десятковою цифрою; б) є двійковою цифрою;
в) є вісімковою цифрою; г) є шістнадцятковою цифрою;
д) малою латинською літерою; е) великою латинською літерою;
ж) латинською літерою; з) латинською літерою або підкресленням.
15. Записати умову того, що до значення змінної **a** можна застосовувати арифметичні операції.
16. Записати умову того, що:
а) значення змінної **b** ділиться націло на **3**;
б) змінну **a** безпечно ділити на змінну **b**;
в) значення цілих змінних **a** і **b** мають однакову парність;
г) значення змінної **a** ділиться на значення змінної **b** без остачі.
17. Написати функцію, яка перевіряє, що два дійсних значення рівні з точністю до ε . Для ε в якості значення за умовчанням використати 10^{-10} .

18. Написати функцію, що повертає ознаку того, що точки площини з координатами (x_1, y_1) , (x_2, y_2) співпадають.

19. Написати функцію, що повертає ознаку того, що точки тривимірного простору з координатами (x_1, y_1, z_1) , (x_2, y_2, z_3) не співпадають.

20. Написати функцію, що повертає ознаку того, що точки з координатами (x_1, y_1) , (x_2, y_2) та (x_3, y_3) лежать на одній прямій.

21. Записати умову того, що відрізки $[a, b]$ та $[c, d]$ осі Ox :

- а) не мають спільних точок;
- б) мають рівно одну спільну точку;
- в) мають нескінченно багато спільних точок.

22. Записати умову того, що довжина відрізка осі Ox з кінцями в точках a і b менше або дорівнює 0,001.

23. Прямую на площині можна задати дійсними коефіцієнтами a, b, c рівняння $ax + by + c = 0$. Написати функцію, що повертає ознаку того, що коефіцієнти дійсно задають пряму.

24. Написати функцію, що повертає ознаку того, що коефіцієнти a, b, c задають:

- а) горизонтальну пряму;
- б) вертикальну пряму;
- в) пряму, що проходить через початок координат.

25. Написати функцію, що за коефіцієнтами a, b, c двох прямих площини повертає ознаку того, що

- а) прямі перпендикулярні;
- б) прямі паралельні, але не співпадають;
- в) прямі колінеарні;
- г) прямі перетинаються.

У цій задачі вважати, що надані коефіцієнти гарантовано задають пряму.

26. Записати умови попередньої задачі у випадку, коли коректність коефіцієнтів прямих не гарантовано.

27. Прямую на площині задано дійсними коефіцієнтами a, b, c рівняння $ax + by + c = 0$. Написати функцію, що повертає ознаку того, що точка з координатами (x, y) лежить на прямій.

28. Прямую на площині задано дійсними коефіцієнтами a, b, c рівняння $ax + by + c = 0$. Написати функцію, що повертає ознаку того, що точки з координатами (x_1, y_1) та (x_2, y_2) лежать в різних півплощинах відносно заданої прямої.

29. Дійсні змінні a, b задають коефіцієнти рівняння $ax + b = 0$. Виразити умову того, що воно:

- а) має єдиний розв'язок;
- б) не має жодного розв'язку;
- в) має нескінченно багато розв'язків.

30. Дійсні змінні a, b, c задають коефіцієнти рівняння $ax^2 + bx + c = 0$. Виразити умову того, що воно:

- а) має рівно два дійсних корені;
- б) має єдиний дійсний корінь;
- в) не має жодного розв'язку;
- г) має нескінченно багато розв'язків.

31. Написати функцію, що задає умову того, що прямокутну цеглину з довжинами ребер a, b, c можна просунути в прямокутне вікно x на y так, щоб її грані були паралельні сторонам вікна.

32. Написати вираз, значенням якого є максимальне зі значень числових змінних a та b .

33. Написати вираз, значенням якого є мінімальне зі значень числових змінних a та b .

34. Написати вираз, що за цілим значенням n повертає значення $(-1)^n$.

35. Що буде надруковано за виконання програми, якщо ввести: а) 0; б) 1; в) 5?

```
a = int(input('Enter an integer: '))
b = 99
if a:
    b = 1
elif a > 3:
    b = 2
else:
```

```
a = int(input('Enter an integer: '))
b = 99
if a > 3:
    b = 1
    a = 1
elif a == 1:
    b = 2
```

```

    b = 3
    print('a =',a, '\nb =', b)

```

a)

```

a = int(input('Enter an integer: '))
b = 99
if a > 3:
    b = 1
if a == 5:
    b = 2
else:
    b = 3
print('a =',a, '\nb =', b)

```

6)
36. Що буде надруковано за виконання програми?

```

def f(a):
    if a > 0:
        return 1
    if a > 4:
        b = 2
    else:
        b = 3
    return b

print(f(0), f(1), f(5))

```

a)

```

def f(a):
    if a > 0:
        return 1
    a = 5
    if a > 4:
        b = 2
    else:
        b = 3
    return b

print(f(0), f(1), f(5))

```

в)

```

else:
    b = 3
    print('a =',a, '\nb =', b)

```

в)

```

a = int(input('Enter an integer: '))
b = 99
if a > 0:
    if a > 4:
        b = 1
    else:
        b = 2
elif a > 3:
    b = 3
elif a > 1:
    b = 4
else:
    b = 5
print('a =',a, '\nb =', b)

```

г)

```

def f(a, b):
    if a > 4:
        return 1
    elif b > 1:
        c = 2
    else:
        c = 3
    return c

print(f(0, 0), f(1, 3), f(5, 0))

```

б)

```

def f(a, b):
    if a > 4:
        return 1
    elif b > 1:
        c = 2
        b = -1
    else:
        c = 3
    return b + c

print(f(0, 0), f(1, 3), f(5, 0))

```

г)

37. Що буде надруковано за виконання програми?

```
def f(n):  
    print('f', end=' ')  
    return n
```

```
a = f(0) and f(2)  
print(a)
```

```
a = f(-3) and f(1)  
print(a)
```

a)

```
def f(n):  
    print('f', end=' ')  
    return n
```

```
a = f(0) or f(2)  
print(a)
```

```
a = f(-3) or f(1)  
print(a)
```

б)

38. Написати функцію з числовим параметром x , яка повертає -1 , 0 , 1 , відповідно, якщо $x < 0$, $x = 0$, $x > 0$.

39. Написати функцію, яка за значеннями дійсних змінних x і y обчислює значення z за формулою

$$z = \begin{cases} x + 2, & \text{якщо } 0 < x \leq 3, \\ x + y, & \text{якщо } x \leq 0, \\ x + y, & \text{якщо } x > 3. \end{cases}$$

40. Написати функцію, яка за коефіцієнтами a і b рівняння $ax + b = 0$ повертає кількість його розв'язків. (Якщо розв'язків нескінченно багато, повертає -1 .)

41. Написати функцію, яка за коефіцієнтами a і b рівняння $ax + b = 0$ повертає його розв'язок. Якщо розв'язок не єдиний, то повертає **None**.

42. Написати функцію, яка за коефіцієнтами a , b , c рівняння $ax^2 + bx + c = 0$ повертає кількість його дійсних розв'язків. (Якщо розв'язків нескінченно багато, повертає -1 .)

43. Написати функцію, яка виводить логічне значення у вигляді **Yes** (для **True**) або **No** (для **False**).

44. Написати функцію, яка за значеннями дійсних змінних x і y обчислює значення $\log_x y$, якщо це можливо, а інакше повертає 0 .

45. Написати функцію, яка за номером місяця повертає пору року.

46. Написати функцію, яка за заданим цілим числом, що задає вік людини, визначає яке зі слів «рік», «роки», «років» треба до нього приписати відповідно до правил української граматики. Наприклад, 21 рік (результат 1), 34 роки (результат 2), 14 років (результат 3).

47. Касиру потрібно видати деяку цілу суму грошей n монетами номіналом по 5 і 2 (яких є необмежений запас) та витратити якомога менше монет. Написати програму, яка обчислює та виводить кількості монет номіналом 5 і 2, які має видати касир. Якщо суму видати неможливо, вивести відповідне повідомлення.

48. Написати функцію, яка за номером дня тижня повертає його назву.

8 Обробка помилок. Винятки

Є дві базові стратегії обробки помилок: помилок можна уникати або можна «ліквідовувати» їх наслідки. Щоб уникнути помилки, треба запрограмувати попередні перевірки та не виконувати те, що не можна. На жаль, не всіх помилок можна уникнути. Зараз розглянемо механізми, які здатні «ліквідовувати» наслідки невдалого виконання операцій.

8.1 Обробка помилкових та неочікуваних ситуацій

Помилкові (англ. *error*) ситуації виникають тоді, коли програма за тих чи інших обставин не може виконуватися за звичайним шляхом виконання. Наприклад, програма повинна просумувати цілі числа, подані у файлі, але не має доступу до вказаного файлу, або виконується цілочислове ділення на нуль. Неочікувані (англ. *unexpected*) ситуації виникають зокрема тоді, коли програма отримує дані, які не знає як інтерпретувати. Наприклад, програма запитує в користувача ввести одну з команд (з наведеного переліку), але користувач вводить щось зовсім інше. В обох випадках звичайне виконання програми стає неможливим.

Виняткова ситуація, або просто **виняток** (англ. *exception*), — це певна ситуація, яка не дозволяє продовжити звичайне виконання програми.

Виникнення виняткової ситуації вимагає спеціальних дій з її розв'язання (*реакції на ситуацію*), тобто оброблення (англ. *handling*). Щоб відокремити оброблення помилок від основної логіки програми, використовують механізм винятків, який підтримується більшістю сучасних об'єктно-орієнтованих мов програмування.

Розрізняють синхронні та асинхронні винятки. **Синхронні винятки** виникають тоді, коли їх кидають у певній частині коду програми за допомогою мовних конструкцій. **Асинхронні винятки** можуть виникнути будь-коли; вони надходять ззовні, наприклад, від апаратного забезпечення, зокрема, процесора у випадку, коли ціле число ділиться на нуль.

У Python винятки генеруються інтерпретатором, коли він бачить помилку часу виконання (англ. *run-time error*), або можуть бути явно згенеровані кодом. Узагалі, для інтерпретатора існує два види помилок: синтаксичні та часу виконання. Синтаксичні помилки виникають тоді, коли оброблюваний інтерпретатором код не задовольняє синтаксису мови Python, наприклад, поява в коді такого рядка

```
x+=1
```

буде діагностована інтерпретатором як синтаксична помилка. Синтаксичні помилки завжди можуть бути перевірені заздалегідь, наприклад, засобами перевірки синтаксичної коректності коду. Помилки часу виконання виникають тоді, коли є проблеми з виконанням синтаксично коректного коду, наприклад, у випадках ділення на 0, розіменування не означеного імені тощо.

За синтаксичних помилок виконання програми завжди зупиняється. У випадку помилок часу виконання генерується виняток і, якщо він не буде оброблений, то виконання програми теж зупиниться. Принципова відмінність цієї ситуації від попередньої у тім, що виняток можна обробити та продовжити виконання програми.

Обробка виняткових ситуацій зазвичай має такий вигляд. За виникнення проблемної ситуації програмні засоби (у тому числі бібліотечні функції та сам код інтерпретатора) або власний код повідомляє про наявність виняткової ситуації. Це називають **киданням винятку** (англ. *raising/throwing an exception*), або **генеруванням винятку**. В іншому місці програми має бути код, що реагує на виняток (цей код описує дії, які слід виконати, якщо щось пішло не так; наприклад, сповіщає про причину неможливості подальшої обробки даних). Його виконання називають **обробленням винятку** (англ. *handling an exception*). Під винятком одночасно розуміють і саму виняткову ситуацію, і значення, яке містить інформацію про неї.

Зауваження щодо термінології. Щодо перехоплення винятку, то англійською це доволі часто звучить як *to catch an exception*, звідси йде те, що винятки *ловлять*, а відповідні частини коду звуть *пастками*. Одночасно з «ловить» (англ. *catch*) вживають термін «обробляє» (англ. *handle*). Щодо літературності, то більш літературним варіантом буде винятки генерувати (а не кидати), перехоплювати та обробляти (а не ловити).

Загально відоме **правило 80/20** каже, що 20% коду забезпечують 80% функціональності програми. Що робить 80% коду? Переважно займається обробкою нетипових ситуацій та помилок ☹. Якщо нічого з цим не робити, то основний код буде настільки захаращений кодом для обробки особливих ситуацій, що стане майже нечитабельним.

Для подолання цієї проблеми використовують різні підходи. Наприклад, відділяють обробку тривіальних випадків від загального випадку, а також намагаються зібрати обробку помилок так, щоб вона не розповзалася по усьому коду.

Зібрати обробку помилок в одному місці програми дозволяє, зокрема, механізм винятків. За появи винятку керування передається обробнику винятку, код якого відокремлений від коду основного випадку. Це порушує структурованість коду, проте, за іншого підходу «розповзання» логіки обробки помилок та неочікуваних ситуацій по різних частинах коду робить код малозрозумілим. Отже, використання винятків є далеко не найгіршим способом створити зрозумілий та гнучкий код.

Сучасні мови програмування мають доволі різний підхід до роботи з винятками. Так, мова C обробляє винятки (сигнали) як асинхронні. У мові C++ використовується синхронна модель винятків, але компілятор може мати перемикач, що дозволить обробляти й асинхронні винятки. Є мови, які намагаються взагалі відмовитися від винятків, пояснюючи це тим, що винятки доволі сильно порушують структурованість коду: за появи винятку буває складно відслідкувати в яку точку коду перейде керування.

Щодо мови Python, то майже всі бібліотечні засоби сповіщають про проблеми часу виконання саме за допомогою винятків. Тому відмовитися від роботи з винятками не вдасться. Водночас існують певні нюанси, які обумовлюють специфічний стиль використання винятків у мові Python.

8.2 Приклад програми з обробленням винятків

Розглянемо задачу обчислення квадратного кореня з дійсного числа, що його вводить користувач. Для неї можливі різні нестандартні ситуації. По-перше, користувач може замість десяткового зображення дійсного числа вести щось інше, наприклад **qwerty**. По-друге, користувач може ввести від'ємне число. Напишемо найпримітивніший код³⁶ для її розв'язання.

```
import math
x = float(input('Enter a real: '))
res = math.sqrt(x)
print(f'{x = }\nresult = {res}')
input('Press Enter to exit')
```

Якщо на запит ввести число набрати послідовність символів, що не утворюють десятковий запис числа, наприклад, **q**, то в підсумку побачимо таке.

```
Enter a real: q
Traceback (most recent call last):
```

³⁶ У цьому текстовому документі задля економії паперу для друку не витримано вимоги PEP8 щодо наявності білих-рядків, що візуально розділяють певні частини коду.

```
File "D:/exc4.py", line 3, in <module>
    x = float(input("Enter a real: "))
ValueError: could not convert string to float: 'q'
```

Під час перетворення рядка на дійсне число виникла помилка, за якою було згенеровано виняток. Ми цей виняток не обробили, тому інтерпретатор аварійно завершив роботу нашої програми та вивів діагностику. Зазвичай інтерпретатор повідомляє поточний стек викликів (тобто які виклики було розпочато, але не завершено на момент появи винятку), поточну виконувану інструкцію, тип винятку та, за наявності, опис проблеми. У нашому прикладі було згенеровано виняток типу **ValueError**. Опис проблеми було згенеровано під час перетворення та він може різнитися для різних інтерпретаторів.

Проведемо ще один експеримент. Спробуємо ввести від'ємне число.

```
Enter a real: -1
Traceback (most recent call last):
  File "D:/exc4.py", line 4, in <module>
    res = math.sqrt(x)
ValueError: math domain error
```

Тепер помилка виникла під час обчислення квадратного кореня. За нею було згенеровано виняток того самого типу **ValueError** і додано опис проблеми.

Змінимо нашу програму так, щоб вона не завершувалася аварійно. Для цього необхідно написати код, що виявляє та обробляє винятки. Також буде цілком доречно основну задачу нашої програми (ввести число, обчислити, вивести результати обчислень) виділити в окрему функцію **calculation**. При цьому у функції обробку помилок передбачати не будемо, натомість винесемо її на рівень програми в цілому.

```
import math
def calculation():
    x = float(input('Enter a real: '))
    res = math.sqrt(x)
    print(f'x = {x}\nresult = {res}')
try:
    calculation()
except BaseException as e: # (1)
    print('Smtg went wrong:', e)

input('Press Enter to exit')
```

Зауваження. Наведений приклад для мови Python є насправді прикладом поганого стилю згідно PEP8 (до порад PEP8 з обробки винятків насправді треба підходити доволі обмірковано, але цей приклад у випадку реального проєкту дійсно варто переписати). Він виключно демонструє обробку винятків та можливості уникнення обробки помилок в основному коді.

Для виявлення та обробки винятків використовується складена інструкція **try**, що в даному прикладі складається з **try**-секції та однієї **except**-секції. Слова **try**, **except** є ключовими.

try-секція має починатися з **try:**, за яким записано набір інструкцій. Про код у **try**-секції кажуть, що він *охороняється* (англ. *guarded section of code*): під час його виконання перевіряється поява винятків. Якщо з'являється виняток, то він може бути перехоплений і оброблений.

Кожна **except**-секція (пастка) починається із ключового слова **except**, далі записують тип винятку (у прикладі **BaseException**); за необхідності доступу до значення винятку вказують ключове слово **as** та ідентифікатор, який буде зв'язано зі значенням винятку. Далі, як зазвичай для складної інструкції, двокрапка та набір інструкцій. Код у пастці призначений для обробки винятків указанного в ній типу та похідних від нього.

Розглянемо, що відбудеться за спроби ввести нечислове значення, скажімо, **q**.

Поява винятка. Під час виконання перетворення до типу **float**, як і раніше, буде згенеровано виняток типу **ValueError**.

Пошук обробника. Після появи винятку починається пошук обробника цього винятку. У процесі пошуку послідовно припиняється виконання всіх інструкцій та викликів функцій доти, доки не буде припинено виконання **try**-секції, для якої існує **except**-секція (пастка), в яку може потрапити виняток зазначеного типу. Клас **ValueError** є похідним від **Exception**, а той у свою чергу від **BaseException**, тому в прикладі обробником буде **except**-секція, позначена (1).

Оброблення винятку. Як тільки відповідну пастку буде знайдено, керування перейде до цієї пастки і виняток буде вважатися *перехопленим*. У цій ситуації інколи кажуть, що було *активовано пастку*. Оскільки для винятку вказано ім'я, то під час входження в пастку буде утворена змінна **e**, значенням якої буде об'єкт винятку. Далі виконуватиметься код пастки, тобто буде виведено наше повідомлення та вміст винятку.

Змінна, що позначає виняток. Вказувати ім'я для винятку, як це було зроблено в прикладі, не є обов'язковим, але може бути корисним, коли треба отримати доступ до значення винятку (наприклад, щоб отримати інформацію щодо виняткової ситуації). У цьому разі виняток є *іменованим*. Змінна, що позначає виняток, припиняє своє існування як тільки завершується виконання пастки незалежно від того, чи існувала вона до спрацьовування пастки.

Продовження виконання. Після виконання пастки решта пасток вже не перевіряється і (за відсутності секції **finally**) буде виконуватися інструкція, наступна за інструкцією **try**, якій належала пастка. У прикладі це виклик функції **input**.

На екрані побачимо таке.

```
Enter a real: q
Smth went wrong: could not convert string to float: 'q'
Press Enter to exit
```

Зауваження. Синтаксис дозволяє записувати пастку відразу для кількох типів винятків, наприклад:

```
except (ValueError, ArithmeticError):
    pass
```

У цьому випадку типи винятків записують в дужках через кому, а під час виконання в пастку потраплятимуть винятки вказаних у переліку типів (та похідних від них).

8.3 Ієрархія вбудованих бібліотечних класів винятків та порядок пасток

Найбільш загальним бібліотечним класом винятків є клас **BaseException**, від якого прямо або непрямо походять інші бібліотечні класи винятків (він є для них *базовим*). Зокрема, його безпосередніми нащадками є класи **Exception**, **KeyboardInterrupt** та деякі інші.

Винятки типу **KeyboardInterrupt** виникають тоді, коли користувач натискає комбінацію клавіш **Ctrl-C**. Переважна більшість вбудованих класів винятків, що генеруються бібліотечними засобами, походить від класу **Exception**. Серед них і згадуваний раніше клас **ValueError**, і деякі інші (див. табл. 20)³⁷.

Табл. 20 — Деякі класи винятків мови Python

Клас винятків	Типова проблема, за якої виникають
ValueError	Невідповідне значення.
ArithmeticError	Арифметична помилка.
ZeroDivisionError	Є похідним від типу ArithmeticError , сигналізує про ділення на нуль.

³⁷ Класів винятків доволі багато, у таблиці наведено дуже малу їх частку.

Клас винятків	Типова проблема, за якої виникають
TypeError	Незастосовність операції до аргументів зазначених типів, наприклад, у випадку спроби просумувати число з рядком.
NameError	Спроба розіменувати неозначене ім'я.

Крім вбудованих класів винятків мова Python дозволяє програмістам означати власні класи та використовувати їх значення в якості значень винятків. Зараз це робитися не буде. Зазначимо лише, що кожен виняток у мові Python має бути екземпляром класу, що є похідним від класу **BaseException**. Власні класи винятків рекомендується будувати як нащадків класу **Exception** (або похідних від нього).

В інструкції **try** одночасно може зазначатися кілька пасток. Під час пошуку необхідної пастки серед пасток інструкції **try** обирається **перша за текстом** пастка, що здатна обробити відповідний тип винятку. Це правило на практиці означає те, що завжди першими слід записувати менш загальні пастки, а більш загальні пастки записувати останніми.

Наприклад, за виконання інструкції

```
try:
    #some action
    pass
except Exception:
    print('Exception')
except ValueError:
    print('ValueError')
print()
```

пастка для винятків типу **ValueError** ніколи не буде активована. Справа в тім, що тип **ValueError** є похідним від **Exception**, тому виняток типу **ValueError** потрапить у пастку для **Exception**, яка записана першою, а після завершення виконання пастки виконання інструкції **try** завершиться і буде виконуватися наступна за нею інструкція – інструкція **print()**.

Отже, пастка для **ValueError** має записуватися до пастки для **Exception**. Аналогічно, пастка для **ZeroDivisionError** має передувати пастці для **ArithmeticError**. Водночас з цим порядок запису пасток для винятків типів **ValueError** та **ArithmeticError** за використання тільки вбудованих класів винятків суттєвим не є.

Пастка для винятків типу **BaseException** завжди має бути останньою: у неї потрапить довільний виняток. Мова Python дозволяє не вказувати тип винятків, які обробляє пастка, наприклад:

```
except:
    # do smth
```

Тоді така пастка без зазначення типу винятків має бути останньою в переліку (принаймні деякі інтерпретатори за цим пильно слідкують). Наведена пастка еквівалентна такій.

```
except BaseException:
    # do smth
```

8.4 Більш точна діагностика помилок

У прикладі з обчисленням квадратного кореня пастка обробляє усі можливі винятки. Неправильно вважати, що це добре. Крім помилок, пов'язаних з обчисленнями вона буде перехоплювати багато чого іншого, наприклад, сигнал про натискання комбінації клавіш **Ctrl-C**, який у великих системах може використовуватися для примусового припинення виконання програми і не завжди бажано обробляти його на загальних засадах. Загальний підхід також ускладнює (а в деяких випадках навіть унеможлиблює) діагностику помилок. Тому

програмувати обробку винятків з однією пасткою для всього, що в неї потрапить, у реальних програмних продуктах не варто.

Як правило, загальні пастки записуються на більш високому рівні: для тих винятків, що пройшли через пастки, записані на більш вкладених рівнях.

```
import math
def calculation():
    try:
        x = float(input('Enter a real: '))
        res = math.sqrt(x)
        print(f'x = {x}\nresult = {res}')
    except Exception as e:
        print('Smth went wrong:', e)
        print('Unable to do work.')

try:
    calculation()
except KeyboardInterrupt:
    print('Aborted.')
except BaseException as e:
    print('\nThere are a global problem here.',e)

input('Press Enter to exit')
```

У цьому прикладі проблеми, пов'язані з обчисленнями, обробляються безпосередньо у функції. На верхньому рівні обробляється ситуація натискання комбінації клавіш **Ctrl-C** та решта винятків, які б потенційно могли б виникнути (або поява яких не була врахована на більш вкладених рівнях).

Узагалі, завжди на верхньому рівні записувати останньою пастку для всіх винятків не така вже й погана річ. У такий спосіб можна уникнути аварійного завершення в тих ситуаціях, коли якісь несподіванки не було передбачені заздалегідь. Наприклад через те, що заздалегідь невідомо, виняток якого типу може бути кинутий. Або за зміни версії Python бібліотечні функції почали кидати інші класи винятків, не передбачені кодом, написаним під старі версії.

Хоча наведена програма вже краща, але недоліки в ній все одно можна знайти: вона не виконує точну діагностику помилок. (Те, що інтерпретатор додає до винятку опис проблеми, є доброю волею інтерпретатора, а зовсім не вимагається мовою.) Щодо цього прикладу, то тут це й не намагалися робити, тому перехопити виняток загального типу цілком достатньо. Але для реальних програмних комплексів однією загальною пасткою для типу **Exception** обмежуватися не слід.

З точки зору тих, хто сповідує ідеологію Python, то значним недоліком коду є те, що одна **try**-секція охороняє код, в різних частинах якого можуть виникнути проблеми різної природи (і, більш того, відповідні винятки будуть мати однаковий тип **ValueError**). PEP8 рекомендує обмежувати кількість коду, що потрапляє в одну **try**-секцію. Притримуючись такого підходу, може виявитися, що інструкції **try** фактично почнуть використовуватися як **if** та їх кількість стане такою, що код втратить читабельність. Про переваги використання винятків та відокремлення коду обробки помилок від основного коду програми в цьому випадку буде вже казати важко.

Не треба маніакально завжди записувати в **try** по одній інструкції. Це не тільки захарастить код, але й значно сповільнить виконання програми. Але якщо в деякі моменти важливо зробити точну діагностику, то й одну інструкцію також можна захистити. У деяких випадках буває доцільно написати власну ієрархію класів винятків та використовувати її, щоб основну обробку

помилки без втрати діагностики винести на більш високий рівень. У довільному випадку слід враховувати вимоги до діагностики помилок та здоровий глузд.

8.5 Загальний вигляд інструкції try та інструкція raise

Інструкція **try** може передбачати не тільки обробку випадків, коли виконання **try**-секції завершилося в результаті появи винятка, але й випадок, коли **try**-секція успішно виконалася до кінця (завершення її виконання відбулося не через появу винятка та не через інструкції **return**, **break**, **continue**). Відповідні інструкції можуть бути записані в секції **else**, яка має розташовуватися після останньої **except**-секції. Також можна записати інструкції, які мають виконатися на завершення виконання інструкції **try** незалежно від того, чи вона була успішно виконана, чи завершилася в результаті появи винятка. Відповідна секція позначається ключовим словом **finally** і має бути останньою секцією інструкції **try**. Секція **finally** не є обробником винятка. Якщо виняток не було перехоплено пасткою, то після виконання секції **finally** продовжується пошук обробника винятка.

Примітка. Якщо в секції **finally** записати інструкції **return**, **break** або **continue**, то за їх виконання виняток буде знищено і пошук його обробника буде припинено. Починаючи з Python 3.14 у такому випадку інтерпретатор генерує попередження.

Також винятки не обов'язково мають генеруватися виключно бібліотечними засобами. Нехай вираз **expr** задає або екземпляр класу, похідного від класу **BaseException**, або клас, похідний від класу **BaseException**. За інструкцією вигляду **raise expr** буде згенеровано виняток. У першому випадку значенням винятку буде значення виразу **expr**. У другому випадку можна вважати³⁸, що значенням винятку буде екземпляр відповідного класу.

Модифікуємо наш приклад. По-перше, припустимо, що треба заборонити користувачу вводити число 13.

³⁸ Насправді це не зовсім так, але ззовні виглядає саме так.

```

import math

def check_input(x):
    if x==13:
        raise ValueError('Not 13 you were asked...')

def calculation():
    try:
        x = float(input('Enter a real, not 13: '))
        check_input(x)
        res = math.sqrt(x)
        print(f'x = {x}\nresult = {res}')
    except Exception as e:
        print('Smth went wrong:', e)
        print('Unable to make work.')
    else:
        print('The work is done.')
    finally:
        print('calculation() is over')

try:
    calculation()
except KeyboardInterrupt:
    print('Aborted.')
except BaseException as e:
    print('\nThere are a global problem here.', e)

input('Press Enter to exit')

```

Якщо ввести число 1, то побачимо таке.

```

Enter a real, not 13: 1
x = 1.0
res = 1.0
The work is done.
calculation() is over
Press Enter to exit

```

Виконання секції **try** завершилося у звичайний спосіб, тому керування переходить секції **else**. За наявності секції **finally** подальше керування переходить секції **finally**, завершення виконання якої завершує виконання всієї інструкції **try**.

Якщо ввести число 13, то побачимо таке.

```

Enter a real, not 13: 13
Smth went wrong: Not 13 you was asked...
Unable to make work.
calculation() is over
Press Enter to exit

```

Виконання секції **try** завершилося появою винятку і потрапили в пастку (2). Після виконання пастки керування також переходить секції **finally**, далі виконання інструкції **try** (1) завершується.

Припустимо, що було натиснуто Ctrl-C. Тоді побачимо таке.

```

Enter a real, not 13: Aborted.
calculation() is over
Press Enter to exit

```

У цьому випадку під час виконання виклику функції було згенеровано виняток типу **KeyboardInterrupt**. За появи винятку виконання **try**-секції інструкції (1) було припинено й розпочато пошук обробника. Незважаючи на те, що обробника в цій інструкції нема, під час припинення виконання інструкції (1) все одно виконалася її секція **finally**, далі було припинено виконання виклику функції, далі припинено виконання **try**-секції інструкції (3) і виняток потрапив у пастку (4), після завершення виконання якої розпочато виконання наступної за **try** інструкції (тобто виклику функції **input**).

Зауваження. Теоретично можна було б об'єднати введення значення із перевіркою його допустимості.

```
def spec_input():
    x = float(input('Enter a real, not 13: '))
    if x == 13:
        raise ValueError('Not 13 you was asked...')
    return x
```

Такий підхід є невдалим, бо призводить до того, що з'являється функція з двома обов'язками: ввести та перевірити, яка обидві дії виконує безпосередньо.

За бажання реалізувати введення «під ключ» можна зробити так

```
def input_float(prompt):
    return float(input(prompt))

def _check_input(x):
    if x == 13:
        raise ValueError('Not 13 you was asked...')

def spec_input():
    x = input_float('Enter a real, not 13: ')
    _check_input(x)
    return x
```

і використати цю функцію

```
def calculation():
    try:
        x = spec_input()
        res = math.sqrt(x)
    # далі без змін
```

За такого виділення обов'язків в окремі функції буде простіше не тільки вносити модифікації в обчислення (за потреби), але й налагоджувати код.

Підсумки щодо синтаксису. Інструкція **try** завжди має починатися з однієї секції **try**. Вона може мати довільну кількість секцій **except**, за якою записується не більше однієї секції **else** і наприкінці не більше однієї секції **finally**. За наявності секції **else** має бути записана хоча б одна секція **except**. Ситуація, коли наявні тільки секції **try** та **finally** є допустимою.

Нюанси щодо виконання 1. Секція **finally** виконується завжди. Навіть у такій ситуації.

```
def f():
    try:
        return 1
    finally:
        return 2
```

За виконання виклику **f()** під час виконання інструкції **return** відбувається завершення виконання інструкції **try**. Тому має виконатися секція **finally**. Вона виконується. З функції повертається те значення, що відповідає останній виконаній інструкції **return**. Тобто функція поверне **2**.

Нюанси щодо виконання 2. Інструкція **try** «охороняє» тільки той код, що записаний у її **try**-секції. Це означає те, що поява неперехоплених винятків у секціях **except**, **else**, **finally** призведе до припинення виконання всієї інструкції **try**.

```
try:    # (1)
    # цей код охороняється інструкцією (1)
    ...
except:
    # цей код не охороняється інструкцією (1)
    ...
else:
    # цей код не охороняється інструкцією (1)
    ...
finally:
    # цей код не охороняється інструкцією (1)
    ...
```

Варіанти запису інструкції *raise*. Інструкцію **raise** можна записати не тільки у вигляді **raise *expr*** :

- у секції³⁹ **except** її можна записати просто як **raise**, за якою поточний виняток буде «перекинуто» далі;
- синтаксис **raise *expr1* from *expr2*** дозволяє не просто кинути виняток ***expr1***, але ще й повідомити про обставини його виникнення: вираз ***expr2*** має задавати виняток⁴⁰, що став причиною.

Зауваження щодо синтаксису 1. У Python 3.11 було введено тип **ExceptionGroup** та секцію **except*** інструкції **try**, які разом забезпечують можливість спрацювання кількох гілок-обробників на одному винятку. Зважаючи на те, що інші сучасні мови програмування таку можливість не забезпечують, тобто це не є спільною рисою багатьох мов програмування, а також зважаючи на те, що семантика цих нововведень трохи уточнювалась у версії 3.11.4 (тобто посеред версії), то в межах вступного курсу програмування не варто розглядати нетипову конструкцію, яка потенційно ще може зазнати змін. Отже, ці можливості не розглядаємо. Відверто кажучи, то навіть секція **else** інструкції **try** в багатьох мовах програмування аналогів не має (і якось же вони існують без неї, причому доволі непогано 😊).

Зауваження щодо синтаксису 2. Python 3.14 секцію вигляду

except (*mun1*, *mun2*, ..., *munN*): набір інструкцій

дозволяє записувати у вигляді

except *mun1*, *mun2*, ..., *munN*: набір інструкцій

8.6 Розмотування стеку

Під час виконання одних інструкцій, наприклад, складених, може розпочинатися виконання інших, *вкладених*, інструкцій та викликів функцій. Як наслідок, виникає допоміжна структура, що зберігає інформацію якнайменш про керування обчисленнями (наприклад, точки повернення з викликів функцій, тобто що слід виконувати по завершенню виконання виклику) та живе за принципом **LIFO** (*Last In First Out*): виконання розпочатої останньою інструкції/виклику завершується першим. Таку структуру прийнято називати **стеком**.

Раніше вже зазначалося, що поява винятку запускає процес послідовного припинення виконання всіх вкладених інструкцій та викликів функцій, виконання яких на поточний момент ще не завершилось. Завершується він тільки тоді, коли буде активовано обробник винятку. Цей

³⁹ За межами **except**-секцій це призведе до появи винятку **RuntimeError**.

⁴⁰ Також можна вказати **None**.

процес називають **розмотуванням стеку** (англ. *stack unwinding*). Під стеком мається на увазі стек інструкцій та викликів, що виконуються.

Важливо. Якщо під час розмотування стеку пастку для винятку не буде знайдено, то програма буде аварійно завершена інтерпретатором з відповідним повідомленням.

8.7 Стратегії обробки помилкових та неочікуваних ситуацій

Якісна програма повинна передбачати можливість надходжень некоректних вхідних даних. За недопустимих даних або помилок у даних завжди бажано про це сигналізувати користувачу, щоб у нього не складалося враження, що програма успішно виконала своє завдання. Повідомлення можуть приймати різний вигляд: або виводитися на консоль/у вікно, або, за консольних застосувань, що передбачають використання в пакетній обробці, програма може завершуватися з повідомленням операційній системі коду помилки. У довільному випадку мовчки нічого за некоректних даних не зробити або, що ще гірше, обробляти некоректні дані як начебто коректні, програма не повинна.

У зв'язку з цим постає питання як організувати обробку помилкових та неочікуваних ситуацій. Способів це зробити доволі багато. Усе залежить від необхідного рівня надійності програми, від специфіки задачі, від структури самого програмного проекту тощо.

Почнемо з функцій. Припустимо, що функція здатна правильно обробляти не всі вхідні дані. Банальний приклад: функція, що виконує обчислення над дійсним аргументом, але при цьому визначена не на всій множині дійсних чисел. Незалежно від того, яка стратегія буде обрана, для функції слід задокументувати:

- область визначення;
- чи перевіряє функція належність аргументів області визначення;
- наслідки застосування функції до аргументів, що не належать області визначення.

Якщо функція не перевіряє коректність аргументів, то поведінка функції на некоректних аргументах може бути:

- невизначеною, тобто розробник не може сказати нічого конкретного або функція поверне якесь сміття, або відомо, що функція може зациклитись;
 - застосовувати таку функцію без попередньої перевірки коректності (або інших гарантій, що аргументи коректні) її аргументів категорично не слід;
- такою, що відомо, що якщо обчислення дійсно неможливі, то функція генеруватиме винятки; типи можливих винятків варто задокументувати;
 - застосовувати таку функцію без попередньої перевірки коректності її аргументів можливо, але треба пам'ятати про необхідність обробки винятків.

Якщо функція перевіряє коректність аргументів, то попередня перевірка в більшості випадків буде надлишковою. За некоректних значень аргументів така функція може:

- повертати якесь заздалегідь встановлене значення, яке відрізняється від значень, що можуть отримуватися за звичайних обчислень; у залежності від конкретної задачі можуть використовуватися такі значення, як **None**, **0**, **-1**, **NaN**;
- генерувати виняток;
- повертати ознаку успішності виконання (**True** якщо все гаразд, інакше **False**);
- повертати код помилки (0 якщо все гаразд, інакше ненульове значення; коди помилок слід задокументувати).

У двох останніх випадках функція проєктується як така, що повертає не тільки результат обчислення, але й ознаку його наявності. Мовою Python така функція може повертати пару значень: ознаку наявності результату обчислень та сам результат.

Універсального рецепту щодо того, коли який підхід використовувати не існує.

Якщо функція виконує математичні обчислення для низки значень певного діапазону, то більш доцільно виконати перевірку коректності діапазону, а у функції перевірок аргументів не виконувати. Якщо функція призначена для разових обчислень упродовж виконання програми, то може виявитись більш зручним виконувати перевірку аргументів усередині функції.

Для функції, що виконує математичні обчислення може бути доцільним мати кілька функцій:

- функцію, що виконує математичні обчислення без перевірки коректності аргументів;
- функцію, що виконує перевірку коректності її аргументів;
- функцію, що виконує математичні обчислення з перевіркою коректності аргументів; ця функція використовує дві попередні.

Обробка помилок у програмі в цілому може виконуватися на різних рівнях коду. Найчастіше виконання кожної задачі, яка може розв'язуватися програмою, оформлюють у вигляді окремої функції та помилки, пов'язані з її виконанням обробляють на рівні цієї функції. На рівні програми в цілому обробляють тільки системні та найбільш критичні помилки та винятки, наприклад, завершення роботи програми, натискання комбінації клавіш **Ctrl-C**, за яких подальша робота програми в цілому вже сенсу не має.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: виняткова ситуація, виняток, синхронні винятки, асинхронні винятки, кидання/генерування винятку, обробка винятку, обробник винятку, пастка, код охороняється, перехоплений виняток, активування пастки, розмотування стеку, **BaseException**, **Exception**, **KeyboardInterrupt**, інструкція **try**, **try**-секція, **else**-секція, **finally**-секція, інструкція **raise**.

2. Яка основна мета використання механізму винятків?
3. Чи можна перехопити та обробити синтаксичні помилки?
4. З яких секцій може складатися інструкція **try**? Яка їх семантика?
5. Що відбувається за генерації винятку?
6. Де припиняється існування змінної, що позначає виняток?
7. Винятки якого типу здатна обробляти пастка?
8. Як вибирається пастка для обробки винятку?
9. У якому порядку слід записувати пастки?
10. Чи обов'язково використовувати іменовані винятки?
11. Який код виконується після виконання коду пастки?
12. Чи може для однієї інструкції **try** виконатися більше однієї пастки?
13. Яка секція інструкції **try** виконується завжди?
14. Чи охороняє інструкція **try** код, що розташований у її секціях **except**, **else**, **finally**?
15. Що відбудеться, якщо пастку для винятку не буде знайдено?
16. Що відбудеться, якщо в секції **finally** записати інструкцію **return**?

Задачі та вправи

1. Що буде надруковано за виконання програми?

```
try:
    print(1, end="")
    print(1 / 0, end="")
    print(2, end="")
except Exception:
    print(3, end="")
except:
```

```
try:
    print(1, end="")
    print(1 / 0, end="")
    print(2, end="")
except Exception:
    print(3, end="")
except
```

```
try:
    print(1, end="")
    print(1 / 0, end="")
    print(2, end="")
except
ZeroDivisionError:
    print(3, end="")
```

<pre> print(4, end="") else: print(5, end="") finally: print(6, end="") </pre> 1)	<pre> ZeroDivisionError: print(4, end="") else: print(5, end="") finally: print(6, end="") </pre> 2)	<pre> except Exception: print(4, end="") else: print(5, end="") finally: print(6, end="") </pre> 3)
<pre> try: print(1, end="") print(1 / 0, end="") except ZeroDivisionError: print(3, end="") except Exception: print(4, end="") else: print(1 / 0, end="") print(5, end="") finally: print(6, end="") </pre> 4)	<pre> try: print(1, end="") print(1 / 0, end="") print(2, end="") except ZeroDivisionError: print(3, end="") except: print(4, end="") else: print(5, end="") finally: print(6, end="") </pre> 5)	<pre> try: print(1, end="") except ZeroDivisionError: print(2, end="") except: print(3, end="") else: print(4, end="") finally: print(5, end="") </pre> 6)
<pre> try: print(1, end="") print(1 / 0, end="") except ZeroDivisionError: print(3, end="") except Exception: print(4, end="") else: print(5, end="") finally: print(2 / 0, end="") print(6, end="") </pre> 7)	<pre> try: print(1, end="") print(1 / 0, end="") except ZeroDivisionError: print(3, end="") except Exception: print(2 / 0, end="") print(4, end="") else: print(5, end="") finally: print(6, end="") </pre> 8)	<pre> try: print(1, end="") print(1 / 0, end="") except ZeroDivisionError: print(2 / 0, end="") print(3, end="") except Exception: print(4, end="") else: print(5, end="") finally: print(6, end="") </pre> 9)
<pre> try: print(1, end="") except ZeroDivisionError: print(3, end="") except Exception: print(4, end="") else: print(1 / 0, end="") print(5, end="") finally: print(6, end="") </pre>	<pre> def f(x): return 1/x try: print(f(0), end="") print(1, end="") except Exception: print(2, end="") else: print(3, end="") finally: print(4, end="") </pre>	<pre> def f(x): y = 1/x print(0, end="") try: f(0) print(1, end="") except Exception: print(2, end="") else: print(3, end="") finally: print(4, end="") </pre>

10)	11)	12)
<pre>def f(x): try: print(int(1 / x), end="") except: print(1, end="") else: print(2, end="") raise TypeError finally: print(3, end="") try: f(0) f(1) except: print(4, end="")</pre>	<pre>def f(x): try: print(int(1 / x), end="") except: print(1, end="") else: print(2,end="") raise TypeError finally: print(3,end="") try: f(1) f(0) except: print(4, end="")</pre>	<pre>def f(x): try: print(int(1 / x), end="") except TypeError: print(1, end="") else: print(2,end="") finally: print(3,end="") try: f(0) f(1) except: print(4, end="")</pre>
13)	14)	15)
<pre>def f(x): try: print(int(1/ x), end="") except TypeError: print(1, end="") else: print(2, end="") finally: print(3, end="") try: f(1) f(0) except: print(4, end="")</pre>	<pre>def f(x): try: return int(1/x) except: print(1,end="") return 0 else: print(2,end="") return 1 finally: print(3,end="") return 2 try: print(f(1) and f(0), end="") print(3, end="") except: print(4, end="")</pre>	<pre>def f(x): try: return int(1/x) except: print(1,end="") return 0 else: print(2,end="") return 1 finally: print(3,end="") try: print(f(1) and f(0), end="") print(3, end="") except: print(4, end="")</pre>
16)	17)	18)

2. Функцію **f** означено так.

```
def f(x):
    try:
        return int(1 / x)
    except TypeError:
        print(1, end="")
    else:
```

```

    print(2, end="")
    return 0
finally:
    print(3, end="")
    return 2

```

Що буде надруковано за виконання?

```

try:
    print(f(0), end="")
    print(f(1), end="")
except:
    print(4, end="")

```

a)

```

try:
    print(f(0) or f(1),
          end="")
    print(3, end="")
except:
    print(4, end="")

```

б)

```

try:
    print(f(1) or f(0),
          end="")
    print(3, end="")
except:
    print(4, end="")

```

в)

3. Функцію **f** означено так.

```

def f(x):
    try:
        return int(1 / x)
    except TypeError:
        print(1, end="")
    else:
        print(2, end="")
        return 0
    finally:
        print(3, end="")

```

Що буде надруковано за виконання?

```

try:
    print(f(0), end="")
    print(f(1), end="")
except:
    print(4, end="")

```

a)

```

try:
    print(f(0) or f(1),
          end="")
    print(3, end="")
except:
    print(4, end="")

```

б)

```

try:
    print(f(1) or f(0),
          end="")
    print(3, end="")
except:
    print(4, end="")

```

в)

4. Написати функцію, яка вводить число в діапазоні **[a; b]**.

5. Написати функцію, яка вводить коефіцієнти квадратного рівняння $ax^2 + bx + c = 0$. (Зверніть увагу: повинна виконуватись умова $a \neq 0$.)

6. Написати функцію, яка вводить коефіцієнти **a, b, c** рівняння прямої $ax + by + c = 0$. (Коефіцієнти **a** й **b** не повинні одночасно бути нульовими.)

7. Модифікувати програму, яка запитує в користувача координати точки дійсної площини, а потім виводить отриману точку, так, щоб вона не тільки використовувала функції, але ще обробляла винятки.

8. Написати програму, яка запитує в користувача коефіцієнти рівняння $ax^2 + bx + c = 0$, а потім виводить отримане рівняння (x^2 можна вивести у вигляді x^2) та його розв'язки. Мають використовуватися функції та оброблятися винятки.

9. Написати функцію, що реалізує введення додатного дійсного числа користувачем.

10. Написати функцію, що приймає два дійсних параметри й повертає результат порівняння їх котангенсів.

11. Написати функцію, що за довжинами сторін трикутника повертає його площу.

9 Організація виконуваного коду в інтерпретаторі

У раніше наведених прикладах функцій зустрічалася ситуація, коли одні ті самі імена в різних контекстах позначали різні змінні. Зараз настав час розібратися із загальними поняттями, що виникають у цьому контексті, а також з тим, як це реалізує інтерпретатор. Останнє питання пов'язане з тим, як інтерпретатор організує код та його виконання, зокрема керування змінними. Почнемо здалеку.

9.1 Область видимості, або область дії означення/оголошення, імені та простір імен

У мовах програмування існують різні механізми та концепції, що дають можливість для різних сутностей використовувати однакові імена.

Історично першим виникло поняття *scope* — область видимості, область дії (означення/оголошення імені). Для окремих мов програмування існують традиції перекладу цього терміну, але по суті в усіх випадках під ним мається на увазі одне те саме. Припустимо, у результаті виконання інструкції було створено змінну **var**. Ті частини коду, де під іменем **var** розуміється саме та змінна, що була створена в цій інструкції, називають *областю видимості*, або *областю дії означення*⁴¹, *імені*, означеного в інструкції.

Конкретні правила визначення області дії означення імені задаються семантикою мови програмування, але найчастіше вона визначається як «певний регіон коду, за винятком його частин, в яких діє інше означення того самого імені» (переприсвоювання значення не рахується); зазвичай ця інша інструкція є вкладеною. Класики теорії програмування для області видимості навіть використовували метафору сиру з дірками: область видимості — це сир, а виключені з неї частини коду — дірки в сирі, де діють інші означення. Зауважимо, що в моменти, коли її означення перекрито іншим, змінна не знищується і продовжує існувати, хоча доступ до її значення є тимчасово втраченим.

Ще один підхід до того, як зробити можливим означення однакових імен в різних (і при цьому не вкладених) частинах коду, — концепція *простору імен* (англ. *namespace*). Простір імен — це певне зібрання імен, усередині якого всі імена мають бути унікальними. При цьому різні простори можуть містити однакові імена і в такому разі ці імена можуть позначати різні сутності. Наприклад, конкретний університет має власний простір назв академічних груп. У межах університету шифр K10 задає конкретну академічну групу і іншої існувати не може. Проте, в іншому університеті теж може існувати група K10 — і це буде вже інша група.

Приклад.

⁴¹ У мовах C/C++ тут буде йтися про область дії оголошення імені, бо відповідна інструкція є інструкцією оголошення. У Python інструкція задає означення, тому область дії означення імені.

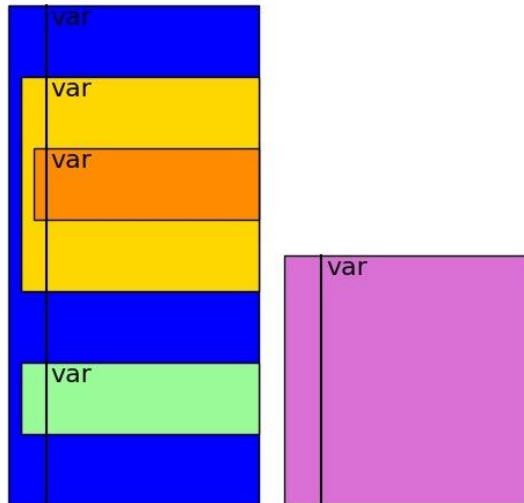


Рис. 12 — Область дії означення змінної та час її існування

На рисунку 12 зображено два програмних модулі, кожен з яких утворює окремий простір імен. У першому є чотири означення змінної **var**, у другому — одне. Кольоровою розміткою на широкій полосі позначено області коду, в яких діють відповідні означення. Ліворуч на вузькій полосі зазначено область коду, під час виконання якої існує змінна, означена у відповідному означенні (при цьому доступ до її значення може бути перекритий іншим означенням). ◀

9.2 Логічна організація виконуваного коду в інтерпретаторі

Інтерпретатор виконує програму не як одне ціле, а певними *блоками коду*. Такі блоки утворюються за кожним файлом з текстом програми мовою Python, що виконується інтерпретатором, за тілом функції, за означенням класу та в деяких інших випадках.

Для інтерпретатора, що працює не в інтерактивному режимі, одиницею вхідного тесту програми є файл з кодом програми мовою Python. Такий файл ми даємо на вхід інтерпретатору, коли запускаємо програму. Такий файл⁴² бере інтерпретатор, коли ми імпортуємо якусь бібліотеку (модуль). З одного боку, модуль — це файл з кодом програми мовою Python. З точки зору інтерпретатора, модуль — це також і внутрішній об'єкт, що відповідає зовнішньому файлу з кодом мовою Python. Різниця в обробці модуля, що імпортується, і оброблюваного інтерпретатором вхідного файлу є, але ці нюанси зараз не мають значення. Документація доволі часто ці випадки не розрізняє, і ми також не будемо.

Блок коду, що утворюється за модулем, вважається блоком верхнього рівня (англ. *top-level*): відповідний текст не міститься в тексті жодного більшого блоку. Блоки коду, що утворюються, наприклад, за тілом означеної у модулі функції або за означенням класу, блоками верхнього рівня не вважаються: їх текст виділяється як частина більшої структурної одиниці тексту; також у деякому розумінні вони є вкладеними блоками.

Приклад.

⁴² Можливо, попередньо оброблений інтерпретатором, але це зовсім не обов'язково.

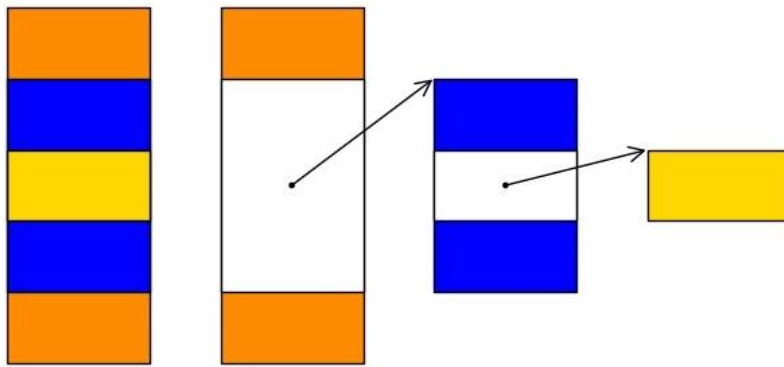


Рис. 13 — Блоки коду

Ліворуч на рисунку 13 схематично зображено вміст текстового файлу з кодом. У ньому наявне означення функції (позначено синім та жовтим), що містить вкладену функцію (позначена жовтим), а також означення змінних та інші інструкції (наприклад, виклики функцій, присвоювання, але не означення функцій); відповідна частина схематично позначена оранжевим кольором.

Такий код буде логічно розбитий на три блоки, що відповідають функціям та глобальній частині тексту. Хоча на рисунку зображено тільки «знання» блоків про блоки, що в них безпосередньо вкладені, але насправді кожен блок знає свій блок верхнього рівня. ◀

Деякі блоки коду виконуються тільки один раз, деякі можуть виконуватися неодноразово (тіло функції). Під час виконання кожного блоку використовуються службові об'єкти — «фрейми виконання» (англ. *execution frame*), що зберігають інформацію про поточну виконувану інструкцію, про означені в блоці змінні тощо. Дуже неформально під виконуваним блоком будемо розуміти блок коду разом із фреймом, в якому він виконується. Така схема дозволяє один той самий блок виконувати неодноразово та паралельно (наприклад, одночасно здійснювати кілька викликів однієї тієї ж самої функції), що може бути корисним і в багатопотоковому програмуванні, і у випадках вкладених (рекурсивних) викликів функцій.

9.3 Робота інтерпретатора зі змінними

Інтерпретатор Python організовує змінні в простори імен, що відповідають блокам коду. Простір імен мови Python можна розуміти як таблицю, що зіставляє імена зі зв'язаними з ними об'єктами.

У кожного **виконаного** блоку коду є його власний **локальний простір імен**, що зберігає означені під час його виконання імена. Відповідні змінні називають **локальними змінними** блоку коду.

Імена, означені за межами означень функцій та класів — на рівні модулів, є **глобальними**; відповідні змінні називають **глобальними**.

Також кожен блок коду «знає» свій глобальний простір імен — простір імен того модуля, в якому розташований його код. Для блоків верхнього рівня їх глобальний простір збігається з локальним.

Під час завантаження інтерпретатор завантажує модуль **builtins** вбудованих функцій, типів тощо і створює відповідний йому простір глобальних імен. Якщо аргументом виклику інтерпретатора є файл з текстом програми, то далі інтерпретатор обробляє цей файл і також створює відповідний йому інший простір глобальних імен. Узагалі, для кожного модуля, що завантажується впродовж роботи програми, створюється його простір імен, що містить глобальні імена модуля.

Як правило, блоки верхнього рівня виконуються один раз, але їх простори імен містять глобальні імена модуля і продовжують існувати навіть після завершення їх виконання.

Код функцій може виконуватися неодноразово. За кожного виклику функції починається виконання відповідного блоку коду і кожного разу створюється новий локальний простір імен. Формальні параметри функції задають імена локальних змінних, що відразу додаються в локальний простір імен, аргументи виклику є значеннями, з якими ці імена зв'язуються. По завершенню виконання локальний простір видаляється разом із змінними.

З цієї схеми стає зрозумілим, чому:

- час існування локальних змінних функції обмежено виконанням її тіла;
- локальні змінні функцій не зберігають свої значення між викликами.

Такі властивості локальних змінних функцій притаманні не тільки мові Python, а майже всім мовам програмування.

- Локальний простір імен у випадку виклику функції прив'язується не до блоку (тексту), а до виконання цього блоку (за допомогою фреймів виконання), тобто до виконуваного блоку. У такий спосіб одночасно можуть виконуватись кілька різних викликів однієї тієї ж самої функції, і для кожного виклику буде існувати свій власний локальний простір імен.

9.4 Розв'язання імен, або Пошук імен

Під час обробки тексту блоку інтерпретатор розрізняє локальні імена блоку та інші імена, що використовуються в блоці. У випадку функції, за відсутності підказок від програміста, локальними іменами її блоку є імена формальних параметрів, а також імена змінних, для яких під час виконання блоку виконуються операції зв'язування імені зі значенням. Ці імена позначають локальні змінні функції.

Крім імен локальних змінних у блоці можуть використовуватись й інші імена (наприклад, глобальні імена). Змінні, які використовуються в блоці і при цьому не є його локальними змінними, у мові Python називають *вільними змінними* (англ. *free variable*) блоку. Вільні змінні можна розіменовувати, але присвоювати їм у загальному випадку не можна.

Приклад 1. Розглянемо функцію (містить помилки!).

```
def f(x):  
    z = 1  
    z = y          #1  
    if False:  
        y = x + a  
    return x + y + z
```

У тілі функції *f* імена *x* (формальний параметр) та *z* (створюється під час виконання функції) є локальними та позначають локальні змінні функції. Ім'я *a* позначає вільну змінну, бо у функції ніде не означається. З цим все прозоро.

Щодо імені *y*, то під час обробки тексту тіла функції інтерпретатор помітив, що існують інструкції, які зв'язують ім'я *y* зі значенням. Тому він відразу зарахував ім'я *y* до локальних імен. За спроби виконати цю функцію виникне помилка на інструкції *z = y*, бо локальна змінна *y* не означена і розіменована бути не може. Щодо того, що насправді інструкція *y = x + a*, в якій означається ім'я *y* ніколи не виконається, то це для класифікації інтерпретатором імені *y* не має жодного значення. ◀

Тепер розглянемо, як інтерпретатор у загальному випадку обробляє імена під час виконання блоку коду.

Випадок 1. Виконується пошук імені, яке інтерпретатор визнав локальним

Пошук відбувається тільки в локальному просторі імен виконуваного блоку. Якщо ім'я не було знайдене і при цьому виконується операція зв'язування імені зі значенням, то в локальний простір імен додається ім'я змінної і подальше виконання відбудеться як зазвичай. Якщо ім'я не

буде знайдено під час розіменування, то буде згенеровано виняток, що може призвести до аварійного завершення виконання програми.

Приклад 2.

```
x = 13

def f(x):
    x = x + 3      # 1
    y = x          # 2
    return x + y   # 3

f(1)
```

У тілі функції імена **x**, **y** є локальними. За виконання інструкції (1) буде змінено значення локальної змінної **x**. За виконання інструкції (2) буде утворено локальну змінну **y**. В інструкції (3) розіменовуються локальні змінні **x**, **y**. ◀

У прикладі 1 під час виконання інструкції (1) виконується спроба розіменувати локальне ім'я **y**. Але локальна змінна **y** на той момент не існує, тому інтерпретатор генерує виняток.

Випадок 2. Виконується пошук імені, яке не визнане інтерпретатором локальним.

У випадку розіменування аналогічно попередньому випадку пошук розпочнеться з локального простору імен поточного виконуваного блоку. Якщо там імені не знайдено, то пошук продовжиться в локальному просторі виконуваного блоку, що охоплює поточний. І так далі, поки не дійде до блока модуля (локальний простір імен цього модуля буде глобальним простором імен поточного виконуваного блоку). Якщо і там ім'я не буде знайдено, то далі буде виконуватися пошук у глобальному просторі імен, що містить вбудовані імена.

Якщо ім'я все одно не буде знайдено, то буде згенеровано виняток. Якщо ім'я знайдено, то результатом його розіменування буде його значення на поточний момент виконання.

Випадок встановлення значення розглядається в наступному підрозділі.

Приклад 3.

```
a = 1
x = 13

def f(x):
    x = x + a      # 1
    y = x
    return x + y

a = 2
f(1)
```

Ім'я **a** у тілі функції позначає вільну змінну (імена **x**, **y** у функції є локальними). У локальному просторі функції воно знайдено не буде. Тому далі пошук продовжиться в оточуючому блоці, що відповідає виконуваному модулю. Там змінна **a** означена і на момент виконання рядка (1) має значення 2. Отже пошук припиниться і в якості значення буде взято значення глобальної змінної **a**. За виконання інструкції (1) локальна змінна **x** отримає значення 3. ◀

Приклад 4. Розглянемо програму.

```

n = 10          # 1
n = 15          # 2

def f():        # 3
    n = 1       # 4
    print("something") # 5

f()             # 6
n = 20          # 7

```

У рядку 1 створюється змінна **n**. Її означення (область видимості її імені) буде діяти в рядках 1, 2, 3, 6, 7. Означення змінної **n** з рядка 4 буде діяти в рядках 4, 5. ◀

У випадку вкладених функцій ситуація має певні нюанси. Для того, щоб мати можливість шукати ім'я в оточуючому блоці, що відповідає тілу функції, то він має виконуватися. Тому виконуваний блок, з якого відбувся виклик вкладеної функції завжди буде відповідати блоку, що охоплює означення цієї вкладеної функції.

Якщо ж функція не є вкладеною в іншу функцію, то охоплюючим для неї є той блок, що містить її означення, а не той блок, що містить її виклик. Отже, для таких функцій пошук вільних імен продовжується в глобальному просторі імен її модуля, модуль, в якому записано її виклик, до уваги не береться.

Приклад 5.

module1.py	module2.py
<pre> n = 1 m = 2 def f(): def g(): n = 3 p = k + m + n n = 2 k = n + m g() print(n, m, k) f() print(n, m) </pre>	<pre> from module1 import f # f позначає ту саму функцію, що й ім'я f у модулі module1. # При цьому глобальні імена f та f знаходяться в різних # просторах. n = 10 m = 20 f() </pre>

Кольоровою розміткою в коді зазначено, де яке означення імені використовується. Блакитні та жовті – глобальні імена, означені в модулях, малинові — локальні імена, означені у функції **f**, зелені – локальні імена, означені у функції **g**.

Ім'я **g** означено тільки під час виконання функції **f**: за межами виклику функції **f** воно використане бути не може. Тому виклик **g** за іменем можливий тільки з виклику **f**. У цьому випадку інтерпретатор «знає», звідки був виклик і для вільних змінних функції **g** спочатку виконує пошук у локальному просторі виконуваного блоку, з якого виконано виклик, – він же є оточуючим блоком.

З глобальними іменами дещо інакше — кожен блок «знає» свій глобальний простір, глобальні простори імен із завершенням виконання блоків зазвичай все одно продовжують існувати. Пошук глобальних імен вже виконується не в просторі блоку, звідки був виклик, а в глобальному просторі того коду, що виконується. Так, незалежно від того, звідки буде викликано функцію **f**, під час її виконання ім'я **m** буде розглядатися як глобальне ім'я з блоку, що відповідає **module1.py**. ◀

У програмуванні, під **розв'язанням імен** (англ. *name resolution*) розуміють процес зіставлення імені значення, тобто те, як відбувається пошук імені транслятором.

9.5 Зміна значень вільних змінних у викликах функцій

Зрозуміло, що функція може змінювати значення власних локальних змінних. Також за певних умов вона може потенційно змінювати значення аргументів виклику, наприклад, якщо вони змінювані. Виникає питання: як реалізувати функцію, що виконує перетворення над вільними змінними? У мові Python відповіддю на це питання є інструкції **global** та **nonlocal**. Обидві інструкції по суті є оголошеннями: вони повідомляють інтерпретатору, як виконувати пошук імені.

Приклад програми з функцією, що змінює глобальні змінні.

```
def f():
    global n
    n = 13

n = 1
f()
print(n)
```

За її виконання буде виведено

```
13
```

Тут використано інструкцію **global**, яка має такий загальний вигляд

global перелік імен через кому

Ця інструкція повідомляє інтерпретатору, що відповідні імена позначають глобальні змінні і пошук у разі встановлення значення має починатися відразу з глобального простору імен. На практиці це означає те, що під час виконання виклику функції присвоювання не призведе до створення локальної змінної, натомість буде виконано присвоювання глобальній змінній.

Інструкція **nonlocal**, яка має такий загальний вигляд

nonlocal перелік імен через кому

повідомляє інтерпретатору, що відповідні імена означені ззовні поточного блоку коду і під час встановлення значення їх слід шукати в блоках, що охоплюють поточний виконуваний блок, але не в глобальному просторі імен. Відповідним змінним також можна присвоювати. Ця інструкція доречна тільки у вкладених функціях.

Інструкції **global** та **nonlocal** мають бути записані в тілі функції до першого використання імені. Документація не фіксує чітко дії інтерпретатора у разі порушення цього правила. Більш того, вона явно попереджає, що наступні реалізації інтерпретатора можуть обробляти цю ситуацію зовсім інакше, що призведе до зміни змісту програми. Отже, нехтувати рекомендацією записувати ці інструкції на початку тіла функції не варто.

Якщо функція змінює значення вільних змінних, то про неї кажуть, що вона має **побічний ефект**. Якщо єдиним призначенням функції є зміна глобальних налаштувань, то ця ситуація ще більш-менш допустима. У решті випадків використання функцій з побічним ефектом вкрай небезпечно, бо зміни глобальних змінних програми перестають бути очевидними і, навіть, не завжди можуть бути прораховані до виконання коду. Крім того побічний ефект ускладнює тестування та наладку коду, бо адекватно протестувати частини коду, що сумісно використовують глобальні змінні, по одинці не вийде.

Не варто змінні програми робити глобальними (хоча це й можливо). Кожна частина програми повинна працювати тільки з тими змінними, які їй необхідні. Це дозволяє уникати їх неправильного використання та незапланованих змін значення (наприклад, коли змінні, що зображають певні сутності, раптом стають допоміжними в обчисленнях).

Використання глобальних змінних у багатопотокових застосуваннях (наприклад, з багатовіконним графічним інтерфейсом) може призводити до неприємних несподіванок, коли різні потоки виконання по черзі почнуть змінювати значення однієї тієї ж змінної.

У сучасному програмуванні рекомендується за можливістю **унікати глобальних змінних**, якщо вони не призначені для обміну даними між потоками виконання, а для обміну даними між функціями одного модуля застосовувати інші засоби (наприклад, передавати як аргументи виклику функції). У великих проєктах глобальні змінні найчастіше призначаються для збереження загальних незмінюваних налаштувань програми.

9.6 Час існування змінних

Інтерпретатор самостійно керує часом існування змінних. Глобальні змінні створюються під час завантаження відповідних модулів, і, за деякими винятками, існують до кінця роботи програми, тобто постійно. Локальні змінні функцій створюються під час виконання виклику функції і по його завершенню автоматично припиняють своє існування (бо знищується локальний простір імен). Змінна, зазначена в заголовку секції **except**, припиняє своє існування по завершенню виконання коду цієї секції.

Природне питання: що відбувається з локальними змінними за появи винятку, коли розпочинається розмотування стеку, тобто пошук обробника і послідовно припиняється виконання вкладених інструкцій та викликів функцій, до того моменту, поки обробник не буде знайдено? Адже в цьому випадку виклик функції завершується доволі специфічно.

Теорія (документація) каже, що за припинення виклику локальний простір імен має знищуватися разом з усіма змінними. Але це ідеальна ситуація. У Tutorial наявна цікава фраза: «Actually, forgetting would be a better way to describe what actually happens.»

Насправді, об'єкт винятку, який передається обробнику, «знає» точку виконання, в якій був створений. Також він «знає» причину та контекст своєї появи. Послідовність винятків можна навіть відслідкувати. На практиці це означає, що до винятку чіпляється службова інформація, яка містить посилання (не обов'язково прямі) на об'єкти локальних просторів імен усіх тих виконуваних блоків, виконання яких було передчасно завершено через появу винятку, також можливе виникнення циклічних посилань. Як наслідок, змінними програміст користуватися вже не може, але відповідні об'єкти можуть зберігатися в пам'яті майже до кінця виконання всієї програми. (Локальні змінні дійсно були знищені, бо була знищена службова змінна, що позначала локальний простір імен, але її значення — відповідний об'єкт, не тільки продовжує існувати, але може передаватися та бути доступним в точці обробки винятку.)

Ця особливість є однією з причин, що в багатьох випадках в Python обробку винятків краще не передавати функціям вищого рівня, а хоча б в мінімальному обсязі виконувати на місці. Особливо, якщо локальні змінні функції володіють ресурсами, які по завершенню виклику треба віддати системі.

9.7 Обробка інтерпретатором функцій: резюме

За виконання інструкції означення функції інтерпретатор:

- створює об'єкт, що зображує функцію в пам'яті, та зв'язує його з іменем функції;
- читає інструкції тіла функції та створює за ними об'єкт, що містить код функції; тіло функції при цьому не виконується;
- за наявності запам'ятовує рядок документації та анотації функції в атрибутах `__doc__` та `__annotations__` об'єкта функції;
- обчислює та запам'ятовує значення аргументів за умовчанням;

– може визначати, які імена в тілі функції дійсно є локальними, а які оголошені як **global** чи **nonlocal** (поки інтерпретатор це робить, але це не означає, що всі майбутні також будуть це робити на етапі обробки означення функції).

Під час виконання виклику функції інтерпретатор:

– знаходить об'єкт, що зберігає функцію, виклик якої здійснюється; якщо об'єкт не знайдено (наприклад, ім'я функції не означено), то виникає помилка;

– обчислює аргументи виклику;

– створює відповідний виклику локальний простір імен блоку функції;

– виконує зв'язування обчислених аргументів виклику з формальними параметрами функції: спочатку обробляються позиційні аргументи, потім ключові. Якщо для формального параметра аргумент не задано, то відповідне ім'я зв'язується з об'єктом, що зберігає значення за умовчанням. Під час виклику кожен формальний параметр має бути зв'язаний рівно з одним аргументом виклику; якщо це не так, то виникає помилка. Імена формальних параметрів задають локальні змінні функції.

– виконує тіло функції;

– за інструкцією **return** завершує виконання виклику функції;

– якщо виконана остання інструкція тіла функції, то також завершує виконання виклику функції;

– визначає значення, що повертається як результат виклику: якщо завершення виконання відбувається за інструкцією **return вираз**, то в якості результату повертається об'єкт, що є значенням виразу; в інших випадках функція повертає **None**;

– завершуючи виклик, знищує локальний простір імен блоку функції (і відповідні змінні) та передає керування в точку виклику;

– завершуючи виклик під час розмотування стеку знищує локальні змінні функції.

Висновки, про які слід пам'ятати:

– час існування локальних змінних функцій обмежено виконанням тіла функції;

– локальні змінні функцій не зберігають свої значення між викликами;

– значення за умовчанням можуть бути змінені впродовж викликів (якщо відповідний тип це допускає).

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: область видимості імені, область дії означення імені, простір імен, локальний простір імен, глобальний простір імен, блок коду верхнього рівня, локальна змінна, глобальна змінна, вільна змінна, побічний ефект функції.

2. Чи можна для різних сутностей використовувати однакові імена? Якщо так, то за яких умов?

3. У яких випадках локальний простір імен збігається з глобальним?

4. Опишіть «життєвий» цикл локального простору імен блоку коду, що відповідає тілу функції.

5. Чи зберігають локальні змінні функцій свої значення між викликами?

6. Як відбувається пошук імен під час виконання?

7. Чи може в тексті програми інструкція виклику функції **f** передувати інструкції означення функції **f**?

8. Чи може функція змінювати свої вільні змінні?

9. У чому суть інструкцій **global** та **nonlocal**?

10. Чи можна записувати інструкції **global** та **nonlocal** посередині функції? Де їх записувати краще?

11. Чому використання глобальних змінних краще мінімізувати?
12. Які проблеми породжує побічний ефект функцій?
13. Як визначається час існування глобальних та локальних змінних?
14. Які проблеми звільнення ресурсів, пов'язані з обробкою винятків, існують у мові Python?

Задачі та вправи

1. Що буде виведено за виконання програми?

<pre>def f(a): a += 1 return a + b def g(x): x = 2 return x a = 3 b = 44 print(f(a), g(b), a, b)</pre> a)	<pre>a = 1 b = 1 def f(a): a += 1 return a def g(x): x = 2 + b return a a = 3 b = 4; print(f(a), g(b), a, b)</pre> б)	<pre>a = 1 b = 1 def f(b): global a a += 1 + b return a def g(x): a = 2 x = a + b return x a = 3 b = 4; print(f(a), g(b), a, b)</pre> в)
<pre>a = 1 b = 2 c = b def f(x, y): x += 1 y += 2 c = a+b return x*y b = 3 print(a, b, c) print(f(a,b), a, b, c)</pre> г)	<pre>a = 1 b = 2 c = b def f(x, y): global c x += 3 y += 2 + c c = a+b+1 return x+y b = 1 print(a, b, c) print(f(a,b), a, b, c)</pre> д)	<pre>a = 1 b = 2 c = b def f(x, y): global c a, b = x, y c = a+b return c b = 3 print(a, b, c) print(f(a,a), a, b, c)</pre> е)

10 Модулі та пакети

У цьому розділі розглядаються елементи роботи з бібліотеками мови Python, переважно на рівні користувача, а не розробника, відповідних засобів. За більш детальною інформацією слід звертатися до офіційної документації на мову Python.

10.1 Модулі

Жодна сучасна програма не складається з одного файлу з текстом програми та не пишеться без використання бібліотек. У випадку мови Python бібліотеки прийнято називати модулями.

Поверхневе знайомство з використанням системних бібліотек відбулося, коли для математичних обчислень використовували функції модуля **math**. Зараз настав час зануритися в роботу з модулями дещо більше.

Модуль (з точки зору файлової системи) є записаним на зовнішньому носії файлом, що містить код мовою Python. Це може бути файл з розширенням **.py** (або **.pyw**), що містить безпосередньо текст програми мовою Python. Це може бути файл з байт-кодом — попередньо обробленим текстом програми мовою Python і записаним на проміжній мові, — що має розширення **.pyc**. Існують модулі, що містяться у двійкових файлах: найчастіше це стандартні бібліотечні модулі, двійковий код яких вбудований в інтерпретатор, але не тільки.

На початку роботи невелика частина системної бібліотеки, наприклад, імена **int**, **float**, **str**, **list**, **tuple**, **print**, **input** тощо, не вимагає попередньої підготовки для використання. На відміну від них решта стандартної бібліотеки та сторонні бібліотеки без попередньої підготовки для використання не доступні. Для того, щоб скористатися бібліотечними засобами, код цих засобів має бути завантажений, а відповідні імена мають бути означені.

З точки зору організації даних у пам'яті, модулі зображуються окремим службовим типом **module** і, як усі сутності Python, зберігаються в об'єктах. Під час завантаження в пам'ять модуля, що зберігається на диску, створюється об'єкт типу **module**, який є образом модуля в пам'яті програми. У цьому об'єкті зберігається така службова інформація, як ім'я модуля, ім'я відповідного файлу (для вбудованих в інтерпретатор модулів може не зазначатися), простір локальних імен модуля (він же простір глобальних імен модуля, тобто, означені в модулі імена) тощо.

У результаті **імпортування** описані в модулі засоби (змінні, функції, типи тощо) стають доступними для використання. Під час імпортування означаються змінні, необхідні для доступу до засобів модуля, а перед цим за потреби модуль з зовнішнього носія завантажується в пам'ять інтерпретатора.

10.1.1 Інструкція імпортування

У найпростішому випадку синтаксис інструкції імпортування такий:

import ім'я модуля

де **ім'я модуля** є ідентифікатором та з точністю до розширення відповідає імені файлу.

Розглянемо, які дії відбуваються під час імпортування. Виконання інструкції імпортування складається з двох основних етапів. На першому етапі за потреби відбувається завантаження модуля в пам'ять інтерпретатора, на другому етапі відбувається означення імен.

Перш за все: інтерпретатор пам'ятає завантажені модулі і, за звичайних обставин, двічі один той самий модуль у пам'ять не завантажує. Імена завантажених у пам'ять модулів (та пакетів) зберігаються в системній змінній **sys.modules**.

Пошук модуля спочатку виконується серед раніше завантажених модулів: якщо його знайдено, то перший етап виконання вважається успішно завершеним. Якщо модуль ще не завантажувався, то розпочинається пошук його коду для завантаження.

Спочатку Python намагається знайти модуль як модуль спеціального типу, якщо він таким не є (не було знайдено), то запускається механізм пошуку звичайних модулів. Останній виконує пошук за шляхами, що містяться в системній змінній **sys.path**. За умовчанням ця змінна містить такі шляхи:

- шлях до папки з виконуваною програмою (у змінній **sys.path** йому може відповідати порожній рядок);
- шляхи, зазначені в системній змінній PYTHONPATH;
- у залежності від способу встановлення Python до змінної **sys.path** можуть додаватися й інші шляхи.

Також значення змінної **sys.path** (вона є списком рядків) можна змінювати під час роботи програми. Але слід пам'ятати, що змінювати значення системних змінних треба вкрай обережно та обмірковано.

Якщо в жодній із зазначених локацій модуль не знайдено, то генерується виняток.

Коли файл знайдено, створюється об'єкт, що зображуватиме модуль у пам'яті, і відбувається вже безпосереднє завантаження модуля в пам'ять, у тому числі, модуль реєструється як вже завантажений.

Далі виконується ініціалізація завантаженого модуля, тобто виконання його коду. Під час цього простір локальних імен модуля поповнюється означеними в модулі іменами — далі ними можна буде користуватися. Зазначимо, що код модуля може містити не тільки означення функцій, змінних, класів: жодних специфічних обмежень на його вміст не існує. Якщо під час ініціалізації генеруються (і не обробляються) винятки, то імпорт модуля вважається неуспішним і відповідне ім'я вилучається з переліку завантажених модулів⁴³.

Отже, після успішного виконання першого етапу імпортування модуль знайдено, завантажено, ініціалізовано, а його ім'я зареєстровано в системному реєстрі завантажених модулів.

На другому етапі виконання імпортування в локальному просторі імен блоку коду, в якому записано інструкцію імпортування, означаються імена. Які саме залежить від того, який вигляд мала інструкція імпортування.

У випадку інструкції імпортування вигляду
`import module1`

у локальному просторі імен буде означено ім'я **module1**, яке буде зв'язано з об'єктом відповідного модуля. Означені в модулі **module1** змінні **n**, **_m** та функція **f** будуть доступні тільки за кваліфікованими іменами, що мають вигляд *ім'я_модуля.ім'я_всередині_модуля*, наприклад, **module1.f**.

Нехай у нас є файл **module1.py**

```
n = 1
def f(): print(n)
_m = 2
f()
```

module1.py

та на виконання запускається така програма.

```
import module1
import module1      # повторного завантаження та виконання модуля не буде
n = 3
module1.f()
print(n, module1.n)
```

Стан пам'яті після виконання цього коду наведений на рис. 14.

⁴³ Подробиці того, чому модуль реєструється до ініціалізації, див. у документації.

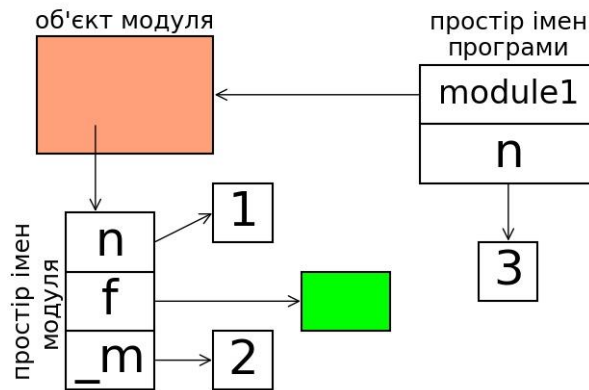


Рис. 14 — Поточний стан пам'яті

Доступ до змінних модуля через кваліфіковані імена дає можливість змінювати значення змінних модуля з коду, розташованого в інших модулях, наприклад, за інструкцією **module1.n = 13** буде змінено значення змінної **n** модуля **module1**. Отриманий у результаті стан пам'яті наведено на рис. 15.

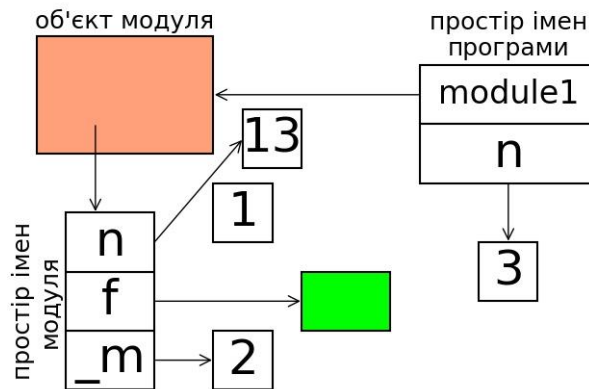


Рис. 15 — Поточний стан пам'яті

У випадку інструкції імпортування вигляду **import ім'я_модуля as ім'я-синонім**, наприклад `import module1 as m`

у локальному просторі імен буде означено ім'я **m**, яке може розглядатися як синонім імені **module1** і буде зв'язано з об'єктом модуля **module1**. Доступ до імен модуля **module1** має відбуватися за кваліфікованим іменем, але з використанням синоніма, наприклад: **m.f**. Використання **module1.f** у даному випадку є помилкою, бо після виконання імпорту модуль позначається іменем **m**, а не **module1**. Стан пам'яті після виконання імпортування наведений на рис. 16.

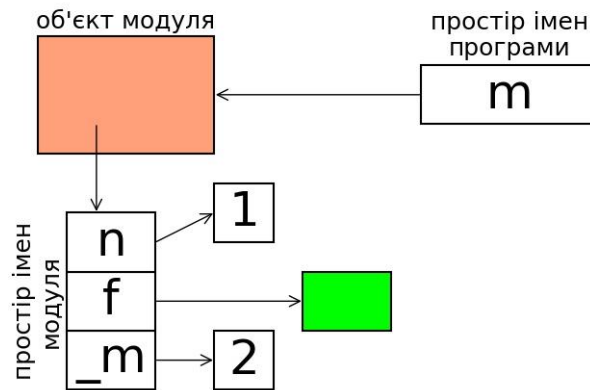


Рис. 16 — Стан пам'яті після виконання імпортування

Інструкція імпортування також може мати вигляд

import перелік імен модулів через кому, можливо, з синонімами

Тоді кожен зазначений модуль обробляється окремо, наче записаний в окремій інструкції.

У випадку інструкції імпортування вигляду

from ім'я модуля import перелік імен через кому

наприклад

```
from module1 import n, f
```

незалежно від того імпортуються всі означені в модулі імена, чи тільки частина, все одно виконується імпорт модуля в цілому (перший етап імпортування). На другому етапі імпортування, коли відбувається означення імен, в локальному просторі імен блоку, в якому записано інструкцію імпорту, означаються імена із зазначеного в інструкції переліку та зв'язуються із значеннями відповідних імен імпортованого модуля. Для наведеного прикладу ім'я **n** буде зв'язано із значенням змінної **n** модуля **module1**. Стан пам'яті після виконання цього імпортування наведений на рис. 17.

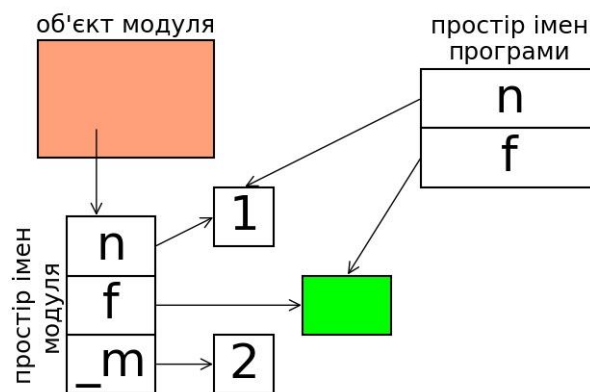


Рис. 17 — Стан пам'яті програми, після виконання програми

З практичної точки зору це означає, що утворена в результаті імпорту змінна **n** та змінна **n** модуля, звідки виконувався імпорт, у початковий момент зв'язані з одним тим самим об'єктом, але під час виконання цілком можлива ситуація, коли вони будуть зв'язані з різними об'єктами і змінюватимуть значення незалежно друг від друга.

За наведеного вище модуля **module1.py** під час виконання коду

```
import module1
from module1 import n    #1
n = 3                    #2
module1.n = 13           #3
print(n == module1.n)
```

після виконання рядка (1) будемо мати стан пам'яті, зображений на рис. 18.

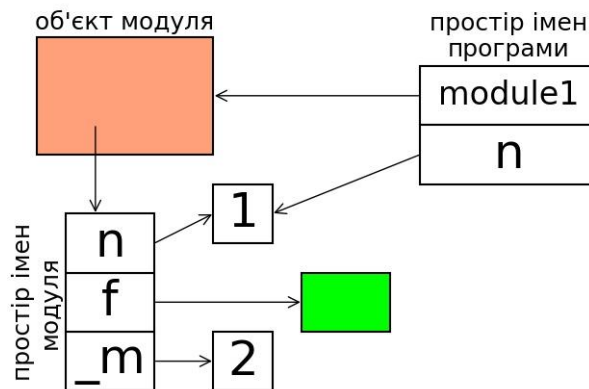


Рис. 18 — Стан пам'яті програми, після виконання програми

після виконання рядка (2) будемо мати стан пам'яті, зображений на рис. 19.

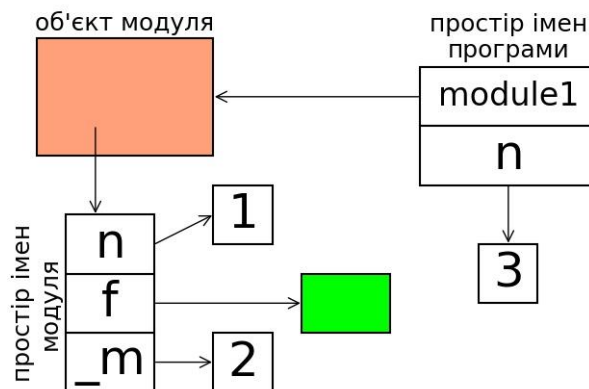
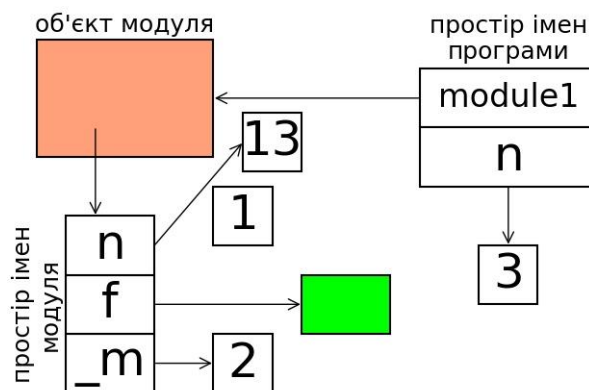


Рис. 19 — Поточний стан пам'яті програми

після виконання рядка (3) будемо мати стан пам'яті, зображений на рис. 20.



за останньою інструкцією буде виведено **False**, бо імена **n** та **module1.n** позначають різні змінні, які змінюють значення незалежно.

Аналогічно імпортуванню вмісту всього модуля, під час імпортування окремих імен можна вказувати для них синоніми, наприклад.

```
from module1 import f as f1
```

У цьому випадку в результаті імпортування в локальному просторі імен утвориться змінна **f1**, яка буде проініціалізована значенням імені **f1** із модуля **module1**. Ім'я **f** у локальному просторі за цією інструкцією означено не буде. Отриманий стан пам'яті зображений на рис. 21.

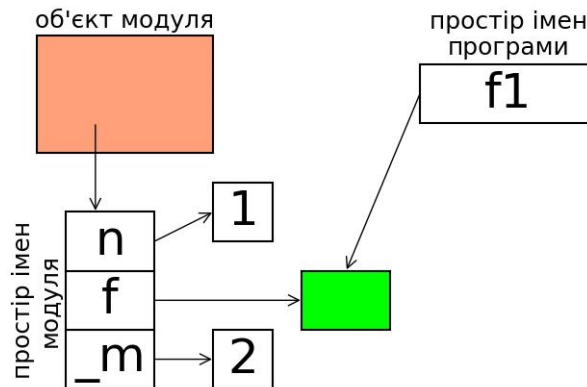


Рис. 21 — Стан пам'яті програми, після виконання програми

Python має ще один вигляд інструкції імпортування,

from ім'я модуля import *

що задає імпортування майже усіх означених у модулі імен в локальний простір імен. У випадку імпортування саме з модуля (а не з пакету) **імпортування всіх імен краще уникати**, щоб уникнути **конфлікту імен** — коли імена, означені в поточному коді, «перекриють» імпортовані імена (або навпаки): в імпортованому модулі може бути означено багато імен, у тому числі ті, що співпадають з іменами, означеними в поточному коді.

За інструкцією вигляду **import *** з модуля будуть імпортовані усі відкриті (англ. *public*) імена модуля, тобто ті, що складають його інтерфейс.

— Якщо в модулі означено змінну **__all__**, то її значенням має бути послідовність рядків, що є означеними⁴⁴ в цьому модулі іменами; усі наявні в **__all__** імена (і тільки вони) вважаються **відкритими** і складають інтерфейс модуля. У цьому випадку з модуля імпортуються тільки зазначені в змінній **__all__** імена.

— Якщо в модулі змінну **__all__** не означено, то відкритими вважаються ті означені в модулі імена, що починаються не з підкреслення. Імена, що починаються з підкреслення, за інструкцією вигляду **from module import *** не імпортуються. Так, за інструкцією **from module1 import *** ім'я **_m** імпортоване не буде.

Імена допоміжних глобальних змінних модуля, що спільно використовуються його частинами (звісно, це дуже погано спроектований модуль!) і не є частиною його інтерфейсу (не призначені для використання ззовні модуля), не повинні бути доступні з-поза його меж.

- **Означені в модулі імена, що не є частиною його інтерфейсу, мають починатися з символу підкреслення!**

⁴⁴ Імена, що набувають значення, у результаті імпортування, також вважаються означеними.

10.1.2 Означені в модулі службові імена

У модулі може бути означена змінна `__all__`, що розглядалася раніше. Крім неї у модулі означаються службові змінні, про деякі з них корисно знати:

`__file__` — містить ім'я файлу з текстом модуля (або з відповідним бінарним файлом, якщо модуль не є вбудованим в інтерпретатор).

Ця змінна дозволяє з'ясувати який саме модуль завантажується, точніше, з якої папки, що може бути дуже корисно, коли існує кілька версій одного модуля, розташованих у різних папках.

`__name__` — містить ім'я завантаженого модуля; для програми в цілому значенням цієї змінної є рядок `__main__`.

Змінну `__name__` найчастіше використовують для того, щоб задати різну поведінку модуля у випадку імпортування як модуля і виконання як основної програми. Наприклад, модуль може містити численні означення функцій або означення класу. Коли він імпортується, нас цікавить доступність означених у ньому засобів. Коли ми запускаємо його як основну програму, то можемо запрограмувати демонстрацію означених у ньому засобів, його тестування тощо. Хоча в останньому випадку слід зауважити, що демо та тестування набагато краще реалізувати в окремому скрипті.

Найчастіше після означень модуля пишуть інструкцію вигляду

```
if __name__ == '__main__':  
    print('модуль запущено як програму')
```

Якщо модуль буде імпортуватися, то повідомлення надруковано не буде: змінна `__name__` буде містити справжнє ім'я модуля (а не ім'я-синонім, якщо імпорт відбувався за допомогою конструкції вигляду `import as`).

Якщо на початку модуля записано рядок документації, то він стає значенням змінної `__doc__`.

10.2 Пакети

Із зростанням розміру програми ми спочатку розкладаємо її на функції⁴⁵; коли стає забагато функцій міркуємо над тим, як їх якнайкраще розподілити по окремих модулях, але настає момент, коли вже й модулі треба теж якось впорядковувати. На цьому етапі виникають *пакети* (англ. *package*), які є зібраннями модулів та вкладених менших пакетів (англ. *subpackage*, дослівно — підпакетів).

У мові Python розглядають пакети двох типів: звичайні пакети (англ. *regular packages*) та пакети простору імен (англ. *namespace packages*).

Історично першими виникли звичайні пакети. Вони чітко прив'язуються до наявної файлової системи, зокрема до структури папок.

Звичайний пакет утворюється за вмістом папки файлової системи, якщо в ній наявний файл `__init__.py` (можливо порожній). Окремі файли (з відповідними мові Python розширеннями), що містяться в цій папці, відповідають модулям, що входять до складу цього пакету. Вкладені папки, що містять файл з іменем `__init__.py`, є вкладеними в нього пакетами.

На відміну від звичайних пакетів пакет простору імен може об'єднувати модулі з різних папок, у тому числі zip-архівів, зокрема «збирати» вміст вкладеного пакету по різних папках.

Наголосимо, що найхарактернішою ознакою пакету простору імен є відсутність файлу `__init__.py` у папці. Отже, якщо кілька записаних в одну папку модулів мають розглядатися як пакет, але файл `__init__.py` відсутній, то відповідний пакет буде оброблятися як пакет простору імен.

⁴⁵ та класи

Під час виконання програми пакети різних типів відрізняються тільки декількома службовими змінними та використовуваними завантажувачами. З точки зору використання означених в них засобів відмінності відсутні.

В якості прикладу розглянемо дуже простий звичайний пакет, що складається рівно з одного модуля. Створимо папку **package1**, в яку покладемо файли **__init__.py** та **m1.py**. У цьому випадку можна розглядати пакет **package1**, що складається з модуля **m1**.

10.2.1 Імпортування пакету

Загальна схема імпортування пакетів аналогічно модулям складається з двох етапів: етапу завантаження та етапу означення імен. Але завантаження пакетів має певні відмінності: спочатку виконується пошук, далі за наявності **__init__.py** виконується його завантаження та його виконання ініціалізує пакет. Так само, як і модулі, усі завантажені пакети реєструються, щоб уникнути їх повторного завантаження та ініціалізації.

У випадку звичайних пакетів під час завантаження завантажується та виконується **__init__.py**. Цей файл надає програмісту доволі широкі можливості керування тим, який саме вміст пакету має імпортуватися. Розглянемо кілька типових випадків.

- Якщо файл **__init__.py** порожній, то він є ознакою того, що інтерпретатор має розглядати пакет як звичайний. Жодних додаткових дій не виконується (до речі, так само як і у випадку пакету простору імен).

- Якщо для правильної роботи модулів пакету необхідно імпортування певних модулів або деяких службових модулів поточного пакету, то відповідні інструкції імпортування доцільно записувати безпосередньо у файлі **__init__.py**. У випадку зміни складу пакету та/або використовуваних у пакеті засобів це дозволяє локалізувати зміни відповідного коду в одній точці.

- Якщо у файлі **__init__.py** явно означено змінну **__all__**, то під час виконання інструкції вигляду **from package import *** будуть імпортовані зазначені в цій змінній імена. У випадку імпортування пакету це можуть бути імена, означені у файлі **__init__.py**, а також імена модулів пакету. В останньому випадку відбувається імпортування модулів.

У загальному випадку синтаксис мови Python на вміст файлу **__init__.py** специфічних обмежень не накладає: в ньому можуть записуватися довільні інструкції, зокрема, створюватися змінні, означатися функції тощо. Але зловживати цим не слід і в реальних проєктах нічого, крім документації (рядок документації утворить значення змінної **__doc__** пакету), інструкцій імпорту та складу інтерфейсу пакету (вміст змінної **__all__**) у цьому файлі краще не записувати.

Модулі та вкладені пакети, що входять до складу пакету, за інструкцією вигляду **import package** імпортуються тільки якщо відповідні інструкції явно зазначено у файлі **__init__.py**.

Приклад. За виконання

```
import numpy
```

буде імпортовано пакет **numpy**. Завдяки виконанню **__init__.py** будуть завантажені допоміжні пакети та модулі та відповідні означення зібрані в локальному просторі модуля (пакета) **numpy**. У точці імпорту буде означено ім'я **numpy**, значенням якого є об'єкт, що зображує в пам'яті завантажений пакет.

За виконання

```
import numpy as np
```

буде виконано те саме, тільки в точці імпорту завантажений пакет позначатиметься іменем **np**.



10.2.2 Імпортування модуля, що входить до складу пакету

Для імпортування модуля або вкладеного пакета, що входить до складу пакета, слід використовувати його повне ім'я. При запису повного імені скісні та зворотні скісні замінюються крапками. Так, для імпортування модуля **m1** можна використати інструкцію.

```
import package1.m1
```

Попереднє імпортування пакету **package1** необхідним не є. За цією інструкцією спочатку буде імпортовано пакет **package1**, а потім модуль **m1** з цього пакету. При цьому для уникнення повторного завантаження та ініціалізації інтерпретатор запам'ятає, що він вже виконував завантаження і **package1**, і **package1.m1**.

Щодо означених імен, то за виконання наведеної інструкції у просторі імен пакету **package1** буде означено ім'я **m1**, значенням якого буде імпортований модуль **m1**. У локальному просторі імен блоку коду, де записано інструкцію імпортування, буде означено змінну **package1**, значенням якої є імпортований пакет. Отримати доступ до імпортованого в такий спосіб модуля можна за використання його кваліфікованого імені **package1.m1**, що складається з імені пакета та імені модуля всередині пакету.

Імпортувати частину пакета можна і з використанням інструкції вигляду.

```
from package1 import m1
```

У цьому випадку так само за потреби буде імпортовано і пакет, і модуль, але в локальному просторі імен з'явиться ім'я **m1**, що позначатиме імпортований модуль.

За інструкцією

```
from package1 import *
```

буде імпортовано пакет, а з нього за відсутності змінної **__all__** усі відкриті імена з **__init__.py**, за наявності цієї змінної – усі зазначені в ній імена. У випадку імпортування з пакету це можуть бути імена, означені у файлі **__init__.py**, або імена модулів пакету. В останньому випадку відбувається імпортування модулів.

Наприкінці зазначимо, що у випадку імпортування пакетів або їх частин в інструкції імпорту за допомогою секції **as** також можна вказувати імена-синоніми.

Приклади.

1.

```
import matplotlib.pyplot
```

Буде імпортовано пакет **matplotlib**, з нього модуль **pyplot**, який у точці імпортування буде доступний за іменем **matplotlib.pyplot**.

2.

```
import matplotlib.pyplot as plt
```

Аналогічно попередньому, тільки в точці імпорту буде означене ім'я **plt**, значенням якого буде імпортований модуль **pyplot**.

3.

```
from matplotlib import pyplot
```

Аналогічно попередньому, тільки в точці імпортування імпортований модуль буде доступний за іменем **pyplot**.

◀

Додаткові синтаксичні можливості. У випадку імпортування частини пакету (окремих модулів, вкладених пакетів) за інструкцією вигляду **from ... import ...** джерело, з якого слід виконати імпортування, може записуватися в абсолютній або відносній (англ. *relative*) формі. (Не дуже формально можна казати про абсолютне та відносне імпортування.) У випадку абсолютної форми вказується повний шлях до відповідного файлу або папки, за виключенням початку, який має бути наявним у змінній **sys.path**. За відносної форми вказується шлях до відповідного дискового файлу або папки відносно поточного модуля, в якому записано інструкцію імпортування. Усі скісні та зворотні скісні, що задають перехід до вкладеної папки,

в обох випадках замінюються на крапки. За відносної форми кожна крапка на початку задає перехід на один рівень вище за ієрархією дискових папок відносно папки поточного модуля (визначається за значенням змінної `__file__`).

10.2.3 Приклади імпортування

Повернемося до нашого пакету **package1**, що складається з модуля **m1**.

```
if (__name__=="__main__"):
    print("module m1 run as main")
else:
    print("module m1 imported")
```

`m1.py`

Розглянемо, що буде виконуватись за імпортування пакету в цілому (див. табл. 21) та за імпортування повного вмісту пакету (див. табл. 22).

Табл. 21 — Імпортування пакету

Вміст <code>__init__.py</code> пакета package1	Результати виконання інструкції import package1
<code>a_package=12</code> <code>print("package1 is imported")</code>	1) На консоль виведено <code>package1 is imported</code> – ознака виконання <code>__init__.py</code> під час імпортування пакету. 2) У локальному просторі імен ім'я package1 позначає імпортований пакет. 3) Змінна a_package в точці імпортування доступна за кваліфікованим іменем package1.a_package .
<code>a_package=12</code> <code>print("package1 is imported")</code> <code>from package1 import m1</code>	1) На консоль виведено <code>package1 is imported</code> <code>module m1 imported</code> Можна бачити, що крім пакету було імпортовано й модуль. 2) У локальному просторі імен ім'я package1 позначає імпортований пакет. У просторі імен пакету означено ім'я m1 , що позначає імпортований модуль. 3) Змінна a_package в точці імпортування пакету доступна за кваліфікованим іменем package1.a_package . Модуль m1 доступний за іменем package1.m1 . Зазначимо, що в цьому прикладі використовувалося абсолютне імпортування.
<code>a_package=12</code> <code>print("package1 is imported")</code> <code>from . import m1</code>	Те саме, що й в попередньому випадку, за винятком того, що тут використовується відносне імпортування.

Розглянемо ще один приклад.

Табл. 22 — Імпортування повного вмісту пакету

Вміст <code>__init__.py</code> пакета package1	Результати виконання інструкції from package1 import *
<pre>a_package=12 print("package1 is imported")</pre>	1) На консоль виведено package1 is imported – ознака виконання <code>__init__.py</code> під час імпортування пакету. 2) До локального простору імен додано ім'я a_package, яке проініціалізовано значенням змінної a_package з пакету package1 (означена в <code>__init__.py</code>). 3) Ім'я package1 у локальному просторі імен не означено.
<pre>__all__=("m1",) a_package=12 print("package1 is imported")</pre>	1) На консоль виведено package1 is imported module m1 imported Можна бачити, що крім пакету було імпортовано й модуль. 2) До локального простору імен додано ім'я m1, що зв'язане з імпортованим з пакету package1 модулем m1. 3) Змінна a_package не імпортована (бо відсутня в <code>__all__</code>). Ім'я package1 у локальному просторі імен також не означено.

Наприкінці, за виконання інструкції **import package1.m1** або **from package1 import m1** незалежно від вмісту `__init__.py` буде імпортовано і пакет, і модуль. Решта вмісту пакету (якщо б він складався з кількох модулів) буде імпортуватися тільки якщо відповідні інструкції зазначені у файлі `__init__.py`.

У першому випадку в локальний простір імен буде додано ім'я пакету **package1**, до модуля **m1** необхідно буде звертатися **package1.m1**. У другому випадку в локальний простір імен буде додано ім'я **m1** і до модуля можна буде звертатися за простим іменем, тобто **m1**.

Більш складні випадки вкладених пакетів та пакетів просторів імен зараз не розглядатимуться, адже для використання бібліотечних пакетів та модулів наведеної інформації більше ніж достатньо.

10.2.4 Отримання довідки за модулями та пакетами

Мова Python має два доволі корисних вбудованих засоби: **help** та **dir**. **help** дозволяє отримати довідку з свого аргументу. Так, за виклику **help(imported_module)** буде виведено довідку за модулем **imported_module**, який попередньо має бути імпортований. Ця довідка буде включати його інтерфейс (відкриті означені в ньому імена), а також рядок документації (якщо він наявний). Аналогічне стосується й пакетів.

Функція **dir** дозволяє отримати перелік означених імен. Так, **dir(imported_module)** поверне список усіх означених в модулі імен (включаючи службові).

10.2.5 Байт-код

Для пришвидшення завантаження Python будує та зберігає на диску **байт-код** модулів. Байт-код є попередньо обробленим текстом програми мовою Python, що його записано на проміжній мові. Виконується байт-код не швидше, а ось завантажується дещо швидше за рахунок

попередньої обробки. Файли з байт-кодом мають розширення **.рус** і зберігаються інтерпретатором у папці **__pycache__**.

Файли з байт-кодом створюються під час завантаження модуля. Далі інтерпретатор запам'ятовує відповідний фал на диску (до імені файлу додаються ще відомості про інтерпретатор та його версію). Також інтерпретатор запам'ятовує стислу інформацію щодо того, щоб у подальшому відслідковувати актуальність байт-коду і за потреби його перебудувати.

Зауваження. Якщо скрипт запускається як основна програма, то збережений байт-код не використовується, натомість завжди виконується повна обробка вхідного тексту.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: модуль, імпортування, інструкція імпортування, завантаження модуля, ініціалізація модуля, змінна **sys.path**, **__all__**, **__name__**, **__file__**, пакет, звичайний пакет, пакет простору імен, **__init__.py**.
2. Що таке модуль з точки зору файлової системи?
3. Що таке модуль з точки зору організації пам'яті програми?
4. Для чого виконується імпортування?
5. Які обмеження накладаються на ім'я модуля? Чи може ім'я модуля містити білі символи? Чи може ім'я модуля починатися з цифри?
6. З яких двох основних етапів складається виконання інструкції імпортування?
7. У яких локаціях відбувається пошук модулів під час імпортування?
8. Чи можна за звичайних обставин під час виконання програми двічі завантажити в пам'ять один той самий модуль?
9. Скільки разів відбувається ініціалізація модуля?
10. Чи імпортується модуль, якщо інструкція імпортування імпортує з нього тільки частину імен?
11. Чи можна вказувати синоніми під час імпортування?
12. Які імена імпортуються з модуля у випадку інструкції вигляду **import *** ?
13. Чому імпортування у вигляді **import *** варто уникати?
14. Як можна задати різну поведінку коду у випадку виконання як основної програми і як модуля?
15. Що таке пакет з точки зору файлової системи?
16. Що таке пакет з точки зору організації пам'яті програми?
17. Які обмеження накладаються на ім'я пакету? Чи може ім'я пакету містити білі символи? Чи може ім'я пакету починатися з цифри?
18. Для чого потрібний файл **__init__.py**? Яким може бути його вміст? Що може бути типовим прикладом його вмісту?
19. Чи можна за звичайних обставин під час виконання програми двічі завантажити в пам'ять один той самий пакет?
20. У чому полягає ініціалізація пакета?
21. Скільки разів відбувається ініціалізація пакета?
22. Чи імпортується пакет, з якого імпортується один з його модулів?
23. Чим значення змінної **__all__** пакету може відрізнитися від значення аналогічної змінної модуля?

11 Рекурсія

Одним зі способів опису об'єктів є рекурсія. Рекурсивними можуть бути правила, що описують структуру виразів деякої мови, означення математичних функцій, алгоритми тощо. Рекурсія є одним з фундаментальних понять програмування й математики. Завдяки їй різноманітним об'єктам можна дати зрозумілий і компактний опис.

Рекурсія виникає тоді, коли явище, або об'єкт, або процес визначається само через себе.

11.1 Рекурсивні означення

Рекурсивне (або інколи кажуть, **індуктивне**) означення задає об'єкт у термінах цього ж об'єкту.

Розглянемо приклади рекурсивних означень об'єктів.

Формула теорії множин. Дамо рекурсивне означення⁴⁶ формули теорії множин (ФТМ), що містить змінні, константи та знаки операцій $\cup$, $\cap$, $\setminus$, $\div$.

1. Окрема змінна є ФТМ. Окрема множина-константа є ФТМ.
2. Якщо F_1, F_2 — ФТМ, то ФТМ є $(F_1 \cup F_2)$, $(F_1 \cap F_2)$, $(F_1 \setminus F_2)$ і $(F_1 \div F_2)$.
3. Інших ФТМ, крім побудованих за правилами 1 і 2, немає.

Приклади ФТМ: $\{1\}$, $(A \cup B)$, $(\{1\} \cap B)$, $((A \cap B) \cup C)$, $((A \cup B) \cup C)$. Зазначимо, що наше означення задає формули теорії множин у так званому повному дужковому записі, коли розставлено всі дужки незважаючи на асоціативність операцій та пріоритети.

Кожне рекурсивне означення повинно мати три частини. Перша частина задає правила будови елементарних (найпростіших) об'єктів. Друга частина задає правила будови більш складних об'єктів з менш складних. Третя частина є заключною і постулює, що все, що не може бути побудоване за правилами, не відповідає означенню. У прикладі ці частини відповідно пронумеровані.

Вираз. Дамо рекурсивне означення виразу, що містить змінні та оператори $*$ та $+$.

1. Окрема змінна є виразом.
- 2.1. Якщо E_1 і E_2 є виразами, то виразами також є $E_1 * E_2$ і $E_1 + E_2$.
- 2.2. Якщо E є виразом, то виразом також є (E) .
3. Інших виразів, крім побудованих за правилами 1, 2.1, 2.2, немає.

За цим означенням виразами є x , y , $x * y$, $x + y$, $(x + y)$, (x) , $x + y + (x) + y$, $((x))$.

Список цілих. Послідовність в інформатиці дуже часто моделюється у вигляді списку (*list*). Дамо рекурсивне означення списку цілих з точки зору математики.

1. $()$ є списком цілих. Такий список не містить жодного елемента; його називають **порожнім**.
2. Якщо n — ціле, l — список цілих, то (n, l) — список цілих. При цьому n називають **головою** (англ. *head*) списку⁴⁷.

3. Інших списків цілих, крім означених за правилами 1 і 2, немає.

Списками цілих за наведеним означенням є: $()$, $(1, ())$, $(2, (1, ()))$.

Рекурсивне означення функції у математиці задає значення функції у певній точці через значення функції у деяких інших точках. Наприклад,

$$n! = \begin{cases} n \cdot (n-1)!, & \text{якщо } n \geq 1, \\ 1, & \text{якщо } n = 0; \end{cases}$$

⁴⁶ Наведене означення не претендує на 100%-у математичну строгість, але не математична строгість є його метою, тому не будемо зараз заходити у формальні хащі.

⁴⁷ Частину l інколи (особливо у функціональних мовах програмування) називають хвостом (tail) списку і кажуть, що список складається з голови та хвоста. У програмних реалізаціях списків в імперативному програмуванні дуже часто хвостом називають останній елемент списку.

$$f(n) = \begin{cases} f(n/2), & \text{якщо } n \text{ парне,} \\ f(3n + 1), & \text{якщо } n \text{ непарне та } n \geq 1, \\ 1, & \text{якщо } n = 1 \end{cases}$$

Примітка. Щодо останньої функції, вік якої більше 50 років, то й досі невідомо, чи є вона всюди визначеною на додатних цілих числах.

У програмуванні рекурсивна функція (не плутати з рекурсивною функцією з теорії алгоритмів!) у своєму тілі містить виклики самої себе; такі виклики навівають **рекурсивними**. Якщо функція викликає себе безпосередньо, то таку рекурсію називають **прямою**. Схематично пряму рекурсію зображено на рис. 22.

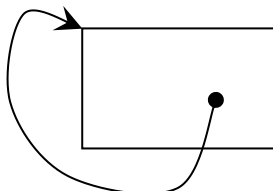


Рис. 22 — Пряма рекурсія

Крім прямої буває й непряма рекурсія, коли функція звертається до себе опосередковано. Відповідні приклади зображено на наступних двох схемах (див. рис. 23 та рис. 24).

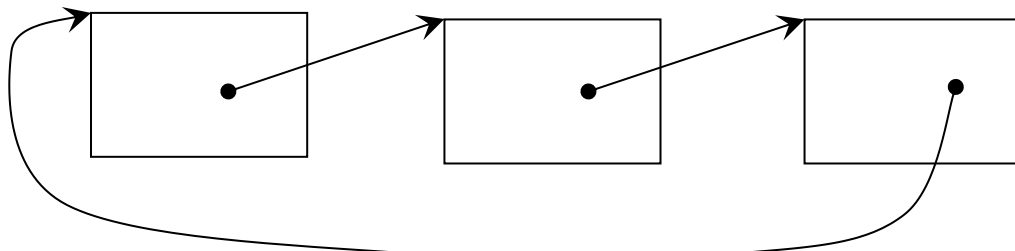


Рис. 23 — Непряма рекурсія

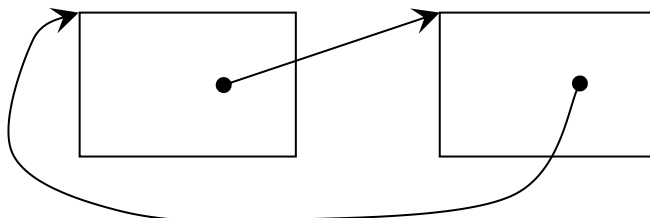


Рис. 24 — Взаємна рекурсія

На рис. 24 зображено так звану **взаємну рекурсію**, коли перша функція викликає другу, а друга — першу.

У всіх випадках важливо, щоб не було зачарованого кола, коли кожен рекурсивний виклик породжує новий рекурсивний виклик. Десь має бути так зване «**дно рекурсії**», на якому рекурсивні виклики припиняються і відбувається повернення з виклику.

11.2 Рекурсивні функції мовою Python та виконання їх викликів

Мова Python дозволяє рекурсивні виклики функцій. Запишемо рекурсивну функцію обчислення факторіала числа. Рекурсивний виклик відмічений червоним.

```
def fact(n: int) -> int:
    """
    Return factorial.

    Key argument:
    n - should be non negative integer, else return 0
    """

    if n < 0:
        return 0
    if n == 0:
        return 1
    return fact(n-1)*n
```

Примітка. Наведену функцію можна пришвидшити, якщо виділити обробку основного випадку ($n \geq 0$) в окрему рекурсивну функцію. Проте, через рекурсію такі обчислення програмувати не прийнято, тому займатись оптимізацією коду тут не будемо.

Виконання викликів нерекурсивних і рекурсивних функцій відрізняється тільки тим, що виконання виклику рекурсивної функції може породжувати інші її виклики (рекурсивні). Схема виконання виклику в обох випадках однакова. Кожен виклик виконується в окремому фреймі, що зберігає необхідну інтерпретатору службову інформацію, тому дані різних викликів, навіть рекурсивних, розділені. Упродовж виконання програми розпочаті та незавершені виклики функцій зберігаються в стеку, що зберігає послідовність вкладених викликів.

Розглянемо процес виконання виклику **fact(2)** з точки зору вмісту стеку. На рисунках 25-30 схематично наведений вміст стеку викликів. Біла область праворуч схематично позначає пам'ять, яку потенційно може зайняти стек викликів.

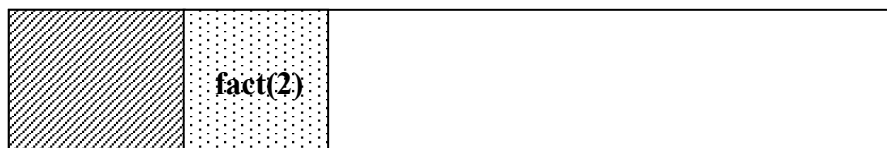


Рис. 25 — Під час виконання виклику **fact(2)**, але перед виконанням рекурсивного виклику

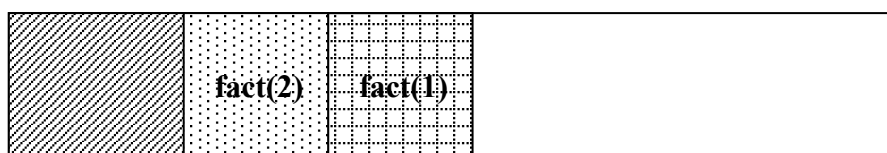


Рис. 26 — Під час виконання виклику **fact(1)**, але перед виконанням наступного рекурсивного виклику

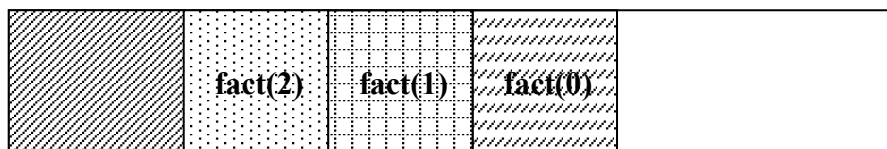


Рис. 27 — Під час виконання виклику **fact(0)**

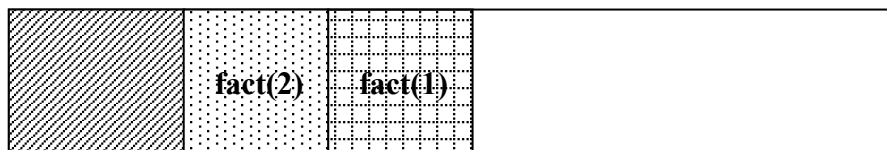


Рис. 28 — Завершення виклику **fact(0)**, продовжується виконання виклику **fact(1)**



Рис. 29 — Завершення виклику `fact(1)`, продовжується виконання виклику `fact(2)`

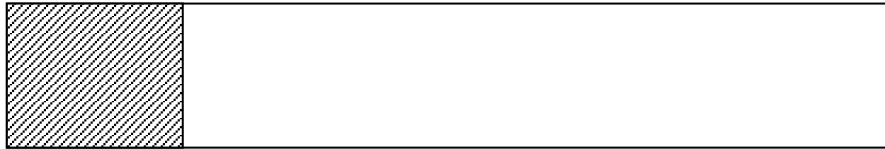


Рис. 30 — Виклик `fact(2)` завершено

З рекурсивними функціями пов'язано два важливих поняття — глибина рекурсії та загальна кількість викликів, породжених викликом рекурсивної функції. Будемо відрізняти глибину рекурсії, на якій перебуває виклик, і глибину рекурсії, породжену викликом.

Глибина рекурсії, на якій перебуває виклик функції, — це кількість рекурсивних викликів, розпочатих і незакінчених у момент початку цього виклику.

Глибина рекурсії, породжена викликом, — це максимальна кількість рекурсивних викликів, які розпочато й не закінчено між його початком та завершенням.

Коли виконується виклик функції, який перебуває на глибині рекурсії m , одночасно існує $m + 1$ екземпляр локальної пам'яті функції. Якщо функція має багато локальних змінних, то за великої глибини вона може аварійно завершитись через нестачу пам'яті.

Загальна кількість рекурсивних викликів, породжених викликом рекурсивної функції, — це кількість рекурсивних викликів, які виконано між його початком і завершенням.

У наведеному прикладі виклику `fact(2)` відповідає глибина рекурсії 2 і загалом відбувається два рекурсивні виклики.

Розглянемо ще один приклад. Напишемо рекурсивну функцію, що обчислює суму квадратів натуральних чисел з множини $\overline{1, n}$.

```
def sum(n):
    if n <= 0:
        return 0
    tmp1 = n*n           # навмисна неоптимальність
    tmp2 = sum(n-1)
    return tmp1 + tmp2
```

Нехай десь у коді наявне таке.

```
N = 800
sumN = sum(N)
```

Під час виконання виклику `sum(N)` виникає рекурсія глибини N (для прикладу це 800), тобто в деякий момент у програмному стеці знаходиться 801 незавершений виклик, кожен з яких потребує виділення пам'яті. Сучасні комп'ютери з їх величезною оперативною пам'яттю мають можливості виконати такий виклик. Але значення N може бути й набагато більшим або функція може мати більше локальних змінних, тоді пам'ять може вичерпатися до того, як почне відбуватися завершення викликів; як наслідок отримаємо аварійне завершення програми.

Глибину рекурсії обмежує не тільки наявний обсяг оперативної пам'яті, але й обмеження на розмір стеку викликів.

Приклад. Нескладний експеримент дозволить оцінити кількість викликів, які одночасно можуть знаходитись у стеку.

```

"""This program roughly estimates possible depth of recursion."""
num = 0

def f1(n):
    global num
    num += 1
    if n > 0:
        f1(n - 1)

try:
    f1(2000)
except RecursionError as e:
    print(f'There were {num} calls, and the program fell down.', e,
          sep='\n')
except BaseException:
    print('Something went wrong...')

```

Отримане значення кількості розпочатих викликів може залежати від інтерпретатора. Виконаний експеримент показує, що воно знаходиться в околі 1000.

Рекурсивні виклики більш притаманні декларативному підходу, коли ми майже не піклуємося про реалізацію (та її ефективність). Як правило, рекурсивні виклики не пришвидшують імперативну програму, а тільки пришвидшують кодування й спрощують сприйняття змісту коду. Рекурсивні функції можуть виявитися непридатними для обробки дуже великих масивів даних. Але є задачі, в яких використання рекурсії допустиме.

- Записуючи рекурсивну функцію, необхідно вміти оцінити можливу глибину рекурсії, розмір пам'яті виклику функції та загальну кількість рекурсивних викликів.

При написанні рекурсивних функцій бажано уникати зайвих змінних (а у випадку Python ще пильнувати за тим, щоб допоміжні локальні змінні не посилались на об'єкти, що далі непотрібні). Такою в нашому прикладі з підсумовуванням є змінні **tmp1**, **tmp2**. Приберемо їх, а також змінимо порядок підсумовування в інструкції **return**, щоб обчислення квадрату відбувалось після завершення рекурсивного виклику (і під час рекурсивних викликів заздалегідь обчислені квадрати не зберігались).

```

def sum(n):
    if n <= 0:
        return 0
    return sum(n-1) + n*n

```

Тепер наша рекурсивна функція стала дещо кращою. Але **обидві рекурсивні функції цього підрозділу можуть бути записані без рекурсії за допомогою циклів** (відповідні цикли наведено серед прикладів розділу 12).

Насправді довільну рекурсію можна замінити *ітерацією* (повторюваними аналогічними діями), тобто *циклами*.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: рекурсивне/індуктивне означення, рекурсивний виклик, пряма рекурсія, непряма рекурсія, взаємна рекурсія, дно рекурсії, глибина рекурсії (на якій перебуває виклик), глибина рекурсії (породжена викликом), загальна кількість рекурсивних викликів.

2. На які характеристики слід звертати увагу за програмування з рекурсивними функціями?
3. Які переваги та які недоліки має програмування з рекурсивними викликами?

Задачі та вправи

1. Написати рекурсивну функцію, яка за заданим n визначає число, що утворюється оберненням десяткового запису n . Наприклад, оберненням 123 є 321, 340 — 43 (незначущі нулі відкидаються).

2. Написати рекурсивну функцію, яка за цілим числом визначає кількість входжень заданої цифри в його десятковий запис.

3. Арифметичні вирази з константами, знаками операцій (+, -, * або /) та дужками опишемо так:

а) дійсна константа є арифметичним виразом;

б) якщо E і F є арифметичними виразами, то $(E+F)$, $(E-F)$, $(E*F)$, (E/F) також є виразами.

в) інших виразів, окрім утворених за правилами пп. а) та б), не існує.

Написати функцію, яка за введеним з клавіатури арифметичним виразом виводить його значення. За один раз користувач вводить або дужку, або оператор, або дійсну константу.

Написати програму, що демонструє роботу функції.

4**. Арифметичні вирази попередньої задачі утворюють повний дужковий запис: на кожний оператор у них наявна пара дужок. Розв'язати попередню задачу, коли повний дужковий запис не вимагається: за відсутності дужок до уваги беруться пріоритети операторів. Вважати, що всі оператори мають властивість лівобічного зв'язування.

5. **Метод половинного ділення.** Нехай наявна функція f мови Python, що обчислює монотонну та неперервну на відрізку $[a, b]$ дійсну функцію f , яка на кінцях відрізка приймає значення різних знаків. Знайти значення $x \in [a, b]$, для якого $f(x) \approx 0$.

6. Написати функцію, яка за цілим числом утворює число, десятковий запис якого утворюється квадратами цифр вхідного числа. Рядки використовувати заборонено. Наприклад, $f(153) = 1259$, $f(-35) = -925$.

7. Написати функцію, яка за невід'ємним цілим числом утворює число, десятковий запис якого утворюється заміною всіх цифр 2 цифрами 5. Рядки використовувати заборонено. Наприклад, $f(1224523) = 1554553$, $f(-25) = -55$.

8. Написати функцію, яка за цілим числом утворює число, десятковий запис якого утворюється викреслюванням кожного другого розряду, починаючи з молодшого. Рядки використовувати заборонено. Наприклад, $f(1234567) = 246$, $f(-1234567) = -246$.

9. Написати функцію, яка за цілим числом утворює число, десятковий запис якого утворюється з кожного другого розряду вхідного числа, починаючи з молодшого. Рядки використовувати заборонено. Наприклад, $f(1234567) = 1357$, $f(-1234567) = -1357$.

10. Написати рекурсивну функцію обчислення числа Фібоначчі. Скільки рекурсивних викликів буде зроблено під час обчислення 20-го числа Фібоначчі? Чи варто було використовувати рекурсію?

11. Запрограмувати з використанням рекурсії обчислення функції f , заданої співвідношеннями

$$f(n) = 1, n \leq 3,$$

$$f(n) = f(n-2) + f(n-3), \neg(n : 2), n \geq 4,$$

$$f(n) = 2f(n/2), n : 2, n \geq 4.$$

12. Модифікувати функцію з попередньої задачі, щоб в результаті її обчислення також обчислювалась загальна кількість рекурсивних викликів та глибина рекурсії.

Обчислити кілька значень. Як багато рекурсивних викликів відбувається? Чи доцільно обчислювати цю функцію за допомогою рекурсії?

12 Ітерація

У перекладі слово «ітерація» означає «повторення». Повторювані аналогічні дії зазвичай програмують з використанням *циклів*.

12.1 Інструкція циклу з передумовою

Запишемо функцію обчислення факторіалу натурального числа без використання рекурсії. За допустимого значення аргументу **n** почнемо зі значення **res** майбутнього факторіалу **1**, яке послідовно помножимо на числа **1, 2, ..., n**. Для цього використаємо лічильник **k**, який буде «проходити» значення **1, 2, ..., n**. Починаючи зі значення **k** рівного **1**, поки **k** менше або дорівнює **n** на кожній ітерації будемо домножувати значення **res** на значення **k**, збільшувати значення **k** і переходити до наступної ітерації. Як тільки **k** стане більшим за **n** ітерації припиняться і значенням **res** буде добуток **1·2·...·n**.

```
def fact_iter(n):
    """
    Return factorial.

    Key argument:
    n - should be non negative integer, else return 0
    """

    if n < 0:
        return 0
    res = 1
    k = 1
    while k <= n:
        res *= k
        k += 1
    return res
```

У наведеному прикладі циклічні, тобто повторювані, обчислення задано за допомогою *інструкції циклу з передумовою* (**while**-інструкції). Вона може мати такий загальний вигляд.

while умова: набір_інструкцій

Слово **while** є зарезервованим, **while умова:** — це *заголовок циклу*, а **набір_інструкцій** — *тіло циклу*.

Інструкція циклу виконується так. Спочатку оцінюється істинність умови в заголовку. Якщо умова істинна, то виконується тіло циклу й знов переходимо до оцінки істинності умови. Якщо вона істинна, все повторюється. Виконання інструкції циклу закінчується, коли значення умови оцінюється як **False**, тобто хибність. Отже, в останньому циклі тільки обчислюється умова, а тіло не виконується. Якщо при першому обчисленні умова хибна, то тіло циклу не виконується жодного разу. Виконання тіла циклу інколи називають *ітерацією циклу*. Інструкції циклу з передумовою відповідає блок-схема, наведена на рис. 31.

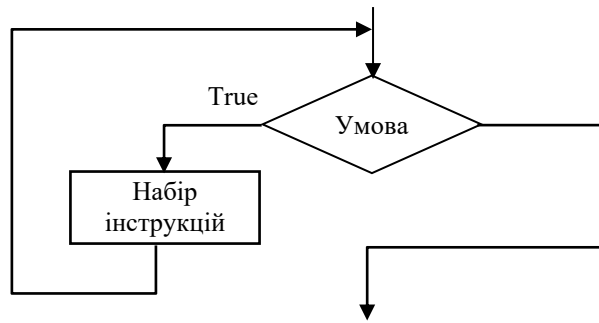


Рис. 31 — Блок-схема інструкції циклу з передумовою

Умову в інструкції циклу називають *умовою продовження*, оскільки, якщо вона істинна, виконання інструкції циклу продовжується. Цикл починається обчисленням умови, тому її ще називають *передумовою*.

Інструкції циклу з передумовою застосовують, як правило, коли кількість повторень циклу наперед невідома і коли кожному (у тому числі й першому) виконанню тіла циклу має передувати перевірка умови.

Приклади

1. За цілим $a \geq 0$ обчислити найменше n , за якого $n! > a$.

Будемо в циклі обчислювати послідовні значення $n!$, поки вони не стануть більше a . Коли цю умову буде порушено, n буде на 1 більше потрібного значення.

```
def min_arg_fact(a):
    """
    Return min {n | n!>a}

    Argument:
    a
    pre: a>=0, else return 0
    """
    if a < 0:
        return 0
    n = 1
    fact = 1          # fact=(n-1)!
    while fact <= a:
        fact *= n
        n += 1
    return n - 1
```

2. Обчислити суму цифр заданого натурального числа.

Щоб знайти суму цифр заданого числа n , можна діяти так. Спочатку сума дорівнює 0. Далі беремо молодшу цифру числа, додаємо її до суми та викреслюємо з числа. Повторюємо ці дії, поки в числі є цифри. Молодша цифра числа n є значенням виразу $n \% 10$, її «викреслення» можна зобразити як $n /= 10$, наприклад, $123 \% 10 = 3$, $123 / 10 = 12$. Оформимо розв'язання функцією з параметром n , яка повертає обчислену суму цифр.

```
def digits_sum(n):
    """
    Sum of number digits.

    Arguments:
    n - number to sum its digits
    pre: n>=0, else return 0
    """
    sum = 0          # сума вже знайдених цифр
    while n > 0:
        sum += n % 10
        n //= 10
    return sum
```

У мовах програмування також зустрічається інструкція циклу, в якій спочатку виконується тіло циклу, а потім перевіряється умова (do-while, repeat-until). Це є інструкція циклу з післяумовою. У мові Python інструкції циклу з післяумовою відсутні. Але це не обмежує можливості мови Python. Без циклів з післяумовою можна обійтися. Кожен цикл з післяумовою **можна замінити циклом з передумовою**

Приклад. Напишемо ігрову програму, яка просить користувача щось ввести, поки він не введе, скажімо, **END**. У цій задачі перше введення має відбутися до перевірки умови завершення (користувач ввів **END**). Цикл з післяумовою міг би бути доречним, але така синтаксична конструкція в Python відсутня. Скористаємося тим, що є, тобто циклом з передумовою. З одного боку, перше введення можна зробити до початку циклу, а потім виконувати в циклі. Але це буде явне дублювання коду, чого варто уникнути. Скористаємося таким прийомом: дамо змінній, що контролює умову, початкове фіктивне значення, за якого цикл гарантовано має виконатися.

```
def main():
    ans = ''
    magic_word='END'
    while ans!= magic_word:
        ans=input(f'If you wanna stop, print {magic_word}: ')
        print('OK. Good bye.')

main()
```

У такого сорту ситуаціях бажано передбачити можливість випадкового введення білих символів на початку або в кінці вхідної послідовності символів. Найпростіший вихід — перейти до рядка, в якому вже прибрано білі символи з початку та з кінця. Цю задачу виконує метод **strip** класу **str**. Перепишемо наш код з урахуванням цієї модифікації: за введеним рядком побудуємо новий, в якому вилучено білі символи на початку та в кінці.

```
def main():
    ans = ''
    magic_word='END'
    while ans!= magic_word:
        ans=input(f'If you wanna stop, print {magic_word}: ').strip()
        print('OK. Good bye.')

main()
```

Тепер, якщо користувач випадково перед **END** введе пробіли або табуляції (на екрані їх можемо не помітити!), то роботу все одно буде припинено.

12.2 Інваріант циклу

«Інваріант» означає «незмінний» (in (*in*) — не, варіант (*variant*) — змінний). Під **інваріантом** циклу розуміють умову, яка істинна на вході в цикл (безпосередньо перед його початком) та в кінці кожної його ітерації.

Інваріанти циклів використовують для доведення коректності алгоритмів. Розглянемо на прикладі коду обчислення факторіалу.

```
res = 1
k = 1
while k <= n:
    res *= k
    k += 1
```

Інваріантом цього циклу є умова **res = (k-1)!**. Дійсно, вона виконується перед початком циклу. Якщо вона виконується на початку ітерації, то вона виконуватиметься й в кінці (це впливає з наведеного коду). Тоді вона виконується й по завершенню циклу. Але по завершенню **k = n+1**, тому **res = n!**. Отже, алгоритм правильно обчислює значення факторіалу.

Для раніше розглянутої функції **min_arg_fact** інваріантом циклу є умова **fact = (n-1)!**, а для функції **digits_sum** обчислення суми цифр натурального числа інваріантом є те, що на **k**-ій ітерації значення змінної **sum** дорівнює сумі **k** молодших цифр числа.

При кодуванні алгоритмів інваріант циклу можна (та іноді дуже варто) записувати в коментарях до циклу.

12.3 Оператор присвоювання

До цього моменту ми розглядали інструкцію присвоювання, яка зв'язувала ім'я зі значенням (об'єктом) та за потреби створювала змінну. Оператор присвоювання **:=** виконує майже те саме, але утворює вираз, значенням якого є присвоєне значення. Завдяки цьому його, на відміну інструкції присвоювання, можна використовувати у виразах. Щодо його реалізації у мові Python, то згідно PEP 572, який увів вирази присвоювання, основним призначенням оператору присвоювання є іменувати значення виразів для їх подальшого використання.

Розглянемо оператор присвоювання на прикладах. Нехай є такий фрагмент коду.

```
import math

x = 1.5
res = math.sin(x) + 1 / math.sin(x)
```

Неважко бачити, що значення **math.sin(x)** обчислюється двічі. Щоб уникнути повторних обчислень, значення треба зберегти в допоміжній змінній. Останню інструкцію замінимо двома такими.

```
tmp = math.sin(x)
res = tmp + 1 / tmp
```

Тепер значення **math.sin(x)** обчислюється один раз.

Той самий ефект буде мати і такий рядок коду, що використовує оператор присвоювання.

```
res = (tmp := math.sin(x)) + 1 / tmp
```

Обчислене значення **math.sin(x)** буде присвоєно імені **tmp** (якщо змінна не існувала, то вона буде утворена; якщо існувала, то значення буде присвоєне існуючій змінній) і далі використане під час обчислення решти виразу. Змінна **tmp**, утворена під час обчислення оператору присвоювання, буде існувати і після завершення обчислення виразу за тими самими правилами, наче відповідна змінна була означена в інструкції присвоювання.

Наведений приклад використання оператора присвоювання не є найбільш типовим. Набагато частіше потреба іменувати значення виникає в обчисленнях умов (та у виразах наповнювання (англ. *comprehension*)).

Доволі часто виникає ситуація, коли в циклі необхідно отримати значення, перевірити його і, якщо все гаразд, то обробити. Відповідний код набуває вигляду.

```
отримати перше значення
while значення підходить:
    обробити значення
    отримати нове значення
```

Написаний за такою схемою код якнайменш буде містити дублювання у частині отримання значення. Природне бажання написати код за дещо іншою схемою.

```
while щойно отримане значення підходить:
    обробити значення
```

Остання схема може бути реалізована за допомогою оператора присвоювання.

```
while (значення:= отримати_значення()) підходить:
    обробити значення
```

Повернемося до прикладу, в якому програма наполегливо просить користувача ввести **END** та перепишемо функцію з використанням оператора присвоювання.

```
def main():
    magic_word='END'
    while (ans:=input(
        f'If you wanna stop, print {magic_word}: ').strip()) != magic_word:
        print('You entered [' , ans, '], the work is going on.', sep='')
    print('OK. Good bye. ')
```

Означення змінної **ans** буде діяти і за межами циклу: від моменту створення змінної і до кінця виконання тіла функції.

Існують синтаксичні обмеження на використання оператора присвоювання. Зокрема ліворуч від оператора **:=** можна записати тільки ідентифікатор (більш складні вирази на відміну від аналогічної інструкції не допускаються), у деяких випадках оператор присвоювання має записуватись тільки в дужках, у деяких випадках він узагалі не дозволений (за більш детальної інформацією варто звертатись до PEP 572).

- Пріоритет оператору присвоювання менший за пріоритети інших операторів, тому в більшості випадків за змістом його все одно треба брати в дужки. Зловживати оператором **:=** не слід.

12.4 Загальний вигляд інструкції циклу **while** у мові Python

Інструкція циклу з передумовою може мати й такий вигляд.

```
while умова: набір_інструкцій_1
else: набір_інструкцій_2
```

У цьому випадку, по аналогії з попереднім, тілом циклу вважається **набір_інструкцій_1**. Сама інструкція виконується майже так само, як і у випадку без секції **else**: оцінюється істинність умови в заголовку; якщо умова істинна, то виконується тіло циклу й знов переходимо до оцінки істинності умови. Якщо вона істинна, все повторюється. Виконання інструкції циклу закінчується, коли значення умови оцінюється як **False**, тобто хибність. У цьому випадку виконується **набір_інструкцій_2**, і тільки після цього виконання циклу завершується. **набір_інструкцій_2** гарантовано буде виконано наприкінці виконання інструкції циклу, навіть якщо тіло циклу не було виконано жодного разу.

Розглянутій інструкції циклу відповідає блок-схема, наведена на рис. 32.

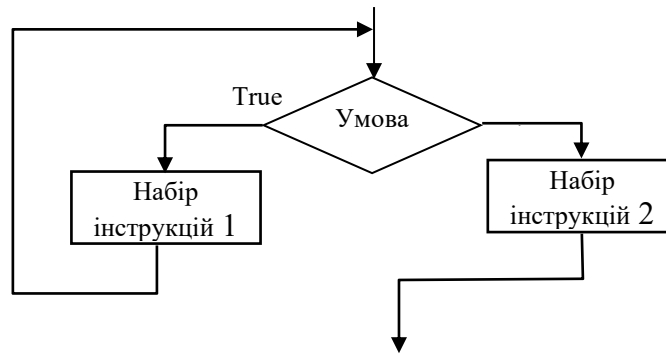


Рис. 32 — Блок-схема інструкції циклу з передумовою за наявності секції else

Приклад. За виконання програми

```

n = 5
k = 1
while k < n:
    print(k, end=', ')
    k += 1
else:
    print(k, end='.\n')
  
```

буде виведено.

```

1, 2, 3, 4, 5.
  
```

12.5 Інструкція циклу for

Повернемося до задачі обчислення факторіалу числа, в якій факторіал обчислювався послідовним множенням початкового значення **1** на числа **1, 2, ..., n**. Подивимося ще раз на частину коду функції **fact_iter**.

```

res = 1      # встановлення початкового значення добутку
k = 1       # встановлення початкового значення множника
while k <= n: # перевірка умови
    res *= k  # основна дія
    k += 1    # перехідна дія перед наступною ітерацією
  
```

Під час виконання цього циклу лічильник циклу – змінна **k**, пробігає послідовні значення від **1** до **n**. Ті самі дії можна записати так.

```

res = 1
for k in range(1, n + 1):
    res *= k
  
```

Інструкція циклу **for**, або **for**-інструкція, у мові Python використовується тоді, коли в циклі слід пройти всі елементи певної сукупності. Вираз **range(1, n+1)** задає послідовність цілих чисел від **1** до **n**.

Дещо узагальнимо. Цикл **for** може мати такий вигляд⁴⁸:

```

for var in вираз: набір_інструкцій_1
або такий
for var in вираз: набір_інструкцій_1
else: набір_інструкцій_2
  
```

де **var** є ідентифікатором, який позначає змінну, що під час виконання циклу пробігає послідовність значень, задану **виразом**. На кожній ітерації значення змінної **var** встановлюється

⁴⁸ Насправді не тільки такий, але зараз більш складні випадки не розглядатимемо.

в черговий елемент послідовності і виконується тіло циклу — **набір_інструкцій_1**. Наприкінці циклу (коли всі елементи послідовності оброблено) виконується **набір_інструкцій_2** (за наявності).

Вираз, що задає послідовність значень, обчислюється один раз на початку виконання циклу. Значення змінної **var** у тілі циклу змінювати можна, але слід розуміти, що ця зміна не вплине на її значення на наступній ітерації. Також змінна **var**, якщо послідовність значень була непорожня, буде існувати й по завершенню циклу (за порожньої послідовності під час виконання циклу вона значення не отримає і, якщо її не було до циклу, то створена не буде).

Приклади.

1. За виконання циклу

```
for k in range(1, 4):
    print(k, end=" ", " ")
    k = 0
else:
    print(4, end=".\n")
```

буде виведено

```
1, 2, 3, 4.
```

2. За виконання циклу

```
for k in range(1, 4):
    print(k, end=" ", " ")
else:
    print(4, end=".\n")
print(k)
```

буде виведено

```
1, 2, 3, 4.
3
```

Можна бачити, що змінна **k** залишає своє останнє значення, яке вона отримувала за виконання циклу. ◀

12.5.1 Обходи

У сучасному високорівневому програмуванні обходи сукупностей значень (особливо, що зберігаються в структурах даних) виконуються за допомогою ітераторів. По суті ітератор є механізмом, що «вміє» поелементно обходити сукупність значень, зокрема «знає» як від поточного елемента перейти до наступного за ним. У зв'язку з цим доволі часто замість «обійти (у циклі) сукупність значень» використовується «проітерувати по сукупності значень». У мові Python інструкція **for** задає саме ітерування.

У наведеному прикладі вираз **range(1, n+1)** задавав сукупність значень, яка має бути пройдена. Специфікою **range** є те, що він задає послідовності цілих чисел. Вираз **range(stop)** задає послідовність цілих від **0** до **stop-1**,⁴⁹ тобто послідовність цілих з інтервалу **[0; stop)**. Вираз **range(start, stop)** задає послідовність цілих **start, start +1, ..., stop -1**. Вираз **range(1, 2)** задаватиме послідовність з одного значення — **1**; вираз **range(1, 1)** задаватиме порожню послідовність.

Вираз **range(start, stop, step)** за додатного значення **step** задає послідовність цілих чисел **start, start+step, start +2·step, ..., start +k·step**, де значення **k** таке, що

start +k·step < stop ≤ start +(k+1)·step.

⁴⁹ Виходячи з особливостей використання, у багатьох мовах програмування діапазони прийнято задавати у вигляді **[from, to)**, тобто ліва межа **from** включається в послідовність, а права межа **to** — ні.

Наприклад, `range(1, 9, 2)` задаватиме послідовність **1, 3, 5, 7**. Як можна бачити, параметр **step** — крок, задає величину зміни цілого значення.

Мова Python допускає від'ємний крок. Вираз `range(start, stop, step)` за від'ємного значення **step** задає послідовність цілих чисел **start, start + step, start + 2·step, ..., start + k·step**, де значення **k** таке, що

$$\text{start} + (k+1) \cdot \text{step} \leq \text{stop} < \text{start} + k \cdot \text{step}.$$

Наприклад, `range(10, 2, -2)` задаватиме послідовність **10, 8, 6, 4**.

Додатково зазначимо, що **range** фіксує не тільки множину значень, але й напрямок її обходу.

Приклади

1. Дуже часто інструкція циклу **for** зустрічається у вигляді

`for k in range(n):` *набір інструкцій*

й задає виконання *набору інструкцій* за значень $k = 0, 1, 2, \dots, n-1$, або у вигляді

`for k in range(1, n+1):` *набір інструкцій*

й задає виконання *набору інструкцій* за значень $k = 1, 2, \dots, n$, або у вигляді

`for k in range(n, 0, -1):` *набір інструкцій*

й задає виконання *набору інструкцій* за значень $k = n, n-1, \dots, 2, 1$. Змінну **k** у цих ситуаціях називають *лічильником циклу*.

2. Функція **print_fraction** друкує перші k дробових знаків десяткового запису дійсного числа v від 0 до 1.

```
def print_fraction(v, k):
    """
    Output k digits after the decimal point.

    Keyword arguments:
    v - real from [0;1)
    k - number of digits to print, k>0
        If not so print nothing.
    """
    if not (0 <= v < 1 and k > 0):
        return
    for i in range(k):
        v *= 10
        d = int(v)
        v -= d
        print(d, end="")
```

Циклом **for** для реалізації чисельних алгоритмів варто користуватися, коли

- 1) кількість ітерацій відома заздалегідь;
- 2) є змінна (яка буде лічильником циклу), що протягом ітерацій пробігає послідовні (або з фіксованим кроком) цілі значення, наприклад, від **1** до **n**, від **0** до **n**, від **1** до **n-1**, від **n** до **1**, від **n-1** до **0** тощо;
- 3) кожна ітерація циклу відбувається з черговим значенням лічильника, усередині ітерації лічильник не змінюється.

12.6 Рекурентні співвідношення в циклічних алгоритмах

12.6.1 Рекурентні співвідношення

Значення факторіалу додатного числа можна задати як число, множене на факторіал **попереднього** числа; при цьому покладається **0! = 1**. Суттєвим тут є те, що насправді визначається послідовність факторіалів **0!, 1!, 2!, ...,** яка визначається двома законами: один

задає перший елемент послідовності, інший каже, як обчислити елемент за його номером та попереднім елементом. Отже, є початкові умови та рекурентне співвідношення, які разом задають послідовність. Для послідовності факторіалів початкові умови — це $0! = 1$, а рекурентне співвідношення — $n! = (n-1)! \cdot n$ за $n > 0$.

Розглянемо загальнішу ситуацію. Припустимо, що перші k елементів послідовності $a_0, a_1, \dots, a_n, \dots$ задано явно (це **початкові умови** $a_0, a_1, \dots, a_{k-1}$), і є закон F , який дозволяє за номером елемента та/або деякими попередніми елементами знайти **всі інші** елементи:

$$a_i = F(i, a_{i-1}, \dots, a_0) \text{ за всіх } i \geq k.$$

Останнє співвідношення називають **рекурентним співвідношенням**. Кожне рекурентне співвідношення разом з початковими умовами визначає в точності одну послідовність.

Приклади

1. Рекурентне співвідношення $a_n = a_{n-1} + d$, $n \geq 1$, з початковими умовами $a_0 = a$ задає арифметичну прогресію з початковим елементом a та кроком d . Співвідношення $a_n = q \cdot a_{n-1}$, $n \geq 1$, та умови $a_0 = a$ визначають геометричну прогресію з початковим елементом a та коефіцієнтом q .

2. Рекурентне співвідношення $S_n = S_{n-1} + \sin(n)$, $n \geq 1$, з початковими умовами $S_0 = 0$ задає суму $\sum_{k=1}^n \sin(k)$.

3. Рекурентне співвідношення $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$, з початковими умовами $F_0 = 0$, $F_1 = 1$ визначає рекурентну послідовність, що має назву **послідовність Фібоначчі**. Вона має такий початок: 0, 1, 1, 2, 3, 5, 8, 13, ... ◀

12.6.2 Програмування циклічних обчислень за рекурентними співвідношеннями

За допомогою рекурентних співвідношень можна розв'язати чимало задач. Головне — знайти рекурентне співвідношення та початкові умови. Далі безпосередньо за ними неважко запрограмувати циклічні обчислення.

Приклад. За цілим числом $n \geq 0$ знайти число Фібоначчі F_n .

Для розв'язання цієї задачі запрограмуємо обчислення за рекурентним співвідношенням $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$, та початковими умовами $F_0 = 0$, $F_1 = 1$.

Якщо $n = 0$, то результатом є 0; якщо $n = 1$, то результатом є 1. За $n \geq 2$ будемо обчислювати елементи послідовності один за одним, доки не отримаємо числа з заданим номером. За F_0 та F_1 обчислимо F_2 , за F_1 та F_2 — F_3 , і так далі. При цьому, щоб обчислити наступний елемент, потрібно знати два останні елементи, що йому передують. Щоб вчасно зупинитися, потрібно також зберігати й щоразу збільшувати номер елемента, обчисленого останнім.

Отже, виділимо поняття: **передостаннє число**, **останнє** та **наступне**, а також **номер** числа, обчисленого останнім. Зобразимо їх змінними, відповідно, **f2**, **f1**, **f** та **i**. Змінні **f2**, **f1** можна розуміти як «віконця», що пересуваються послідовністю Фібоначчі.

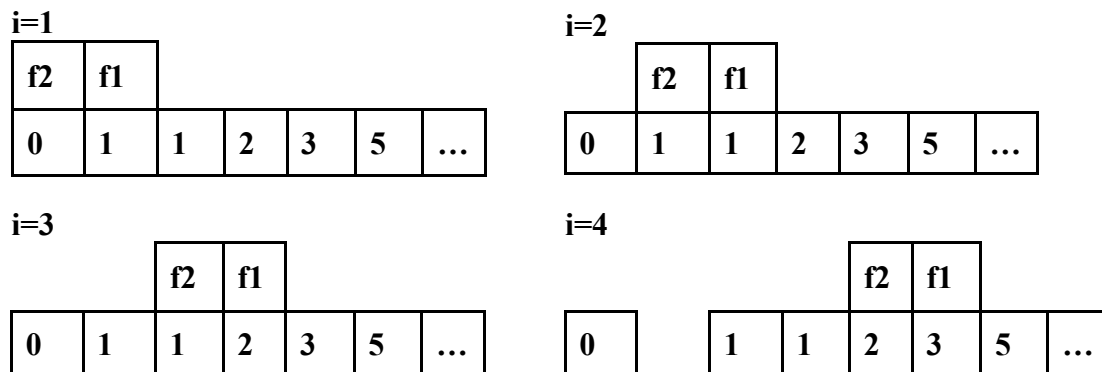


Рис. 33 — Схема обчислення елементів послідовності Фібоначчі

Отже, зі збільшенням i віконце пересувається «на одну позицію праворуч» (див. рис. 33, 34). Для цього обчислюється наступний елемент і зберігається в змінній f , значення $f1$ стає значенням $f2$, а f — значенням $f1$.



Реалізуємо наведені міркування у функції.

```
def fibo(n):
    """
    Return the n-th Fibonacci number.

    pre: n>=0, else return -1
    """
    if n < 0: return -1
    f2, f1 = 0, 1
    if n == 0: return f2
    for i in range(1, n):      # f1=F_i
        f = f1 + f2
        f2 = f1; f1 = f      # зсув «віконця»
    return f1
```

У коментарі до заголовку циклу зазначено його інваріант: значенням змінної $f1$ є i -е число Фібоначчі F_i .

Увага! Під час «зсуву віконця» першою нове значення має отримувати його «найстаріша» змінна, тобто спочатку $f2$, а потім $f1$. Присвоювання $f1 = f$; $f2 = f1$ були б дуже грубою помилкою.

Для перевірки цієї функції потрібно, наприклад, виконати обчислення для аргументів $-1, 0, 1, 2$ та деяких інших (вибрати значення, за якого тіло циклу виконується більше одного разу, скажімо, 5) так, щоб було перевірено обробку некоректних параметрів, обчислення початкових елементів, одно- та багаторазове виконання тіла циклу.

Якщо використати специфіку присвоювань мови Python, то тіло циклу можна записати зовсім коротко.

```
def fibo(n):
    """
    Return the n-th Fibonacci number.

    pre: n>=0, else return -1
    """
    if n < 0: return -1
    f2, f1 = 0, 1
    if n == 0: return f2
    for i in range(1, n):      # f1=F_i
        f2, f1 = f1, f1 + f2  # зсув «віконця»
    return f1
```

Щодо ефективності останнього варіанту присвоювань, то за залежності поточного елемента послідовності не від двох попередніх, а, скажімо, від 100 попередніх елементів, присвоювання кільком змінним одночасно може вимагати більше прихованих додаткових дій із збереження

проміжних значень, через що буде виконуватися більш повільно. Але воно насправді може залежати і від конкретної версії інтерпретатора; який варіант буде швидшим (з присвоюванням списку або з кількома присвоюваннями) варто перевіряти експериментальним методом. ◀

Зверніть увагу: щоб знайти n -й елемент послідовності, заданої рекурентним співвідношенням, необхідно послідовно обчислювати 1-й, 2-й, ..., n -й елементи. Але за використання рекурсивних викликів кількість обчислювальних дій може бути набагато більшою!

Рекурентні співвідношення використовують у програмуванні обчислень, пов'язаних з послідовностями. Співвідношення дозволяють формально виразити залежності між величинами, які обчислюються послідовно, а на основі цих виразів легко запрограмувати цикл. Для розв'язання задачі треба:

- а) зрозуміти, яку послідовність утворюють величини;
- б) записати відповідне рекурентне співвідношення;
- в) визначити перші члени послідовності, які обчислюються без співвідношення (за початковими умовами);
- г) сформулювати умову продовження, за якої застосовується співвідношення.

Після цього згадані вище «деталі конструктора» й порядок їх розташування в алгоритмі стають очевидними.

У кінці протестувати отриманий код та за потреби виправити арифметичні помилки в початковій ініціалізації та умові продовження циклу.

Розглянемо приклади, в яких рекурентні співвідношення не вказано явно.

Приклад. За натуральним n обчислити значення виразу $\sqrt{2 + \sqrt{2 + \dots + \sqrt{2}}}$ (n знаків кореня).

Помітимо: результат можна обчислити як n -й елемент послідовності $\sqrt{2}$, $\sqrt{2 + \sqrt{2}}$, $\sqrt{2 + \sqrt{2 + \sqrt{2}}}$, ... Нехай $a_n = \sqrt{2 + \sqrt{2 + \dots + \sqrt{2}}}$ (n знаків кореня).

Щоб розв'язати цю задачу, знайдемо для послідовності (a_n) рекурентне співвідношення та початкові умови. На кожному кроці до попереднього результату додається 2 й береться квадратний корінь з цієї суми, отже, $a_n = \sqrt{2 + a_{n-1}}$ (це наше рекурентне співвідношення). Помітимо, що $a_1 = \sqrt{2} = \sqrt{2 + 0}$. Тому для систематичності покладемо $a_0 = 0$ (це наші початкові умови). Отже, щоб обчислити a_n , достатньо, починаючи з a_0 , n разів виконати обчислення за рекурентним співвідношенням. Можна програмувати.

```
ai = 0
for _ in range(n):
    ai = math.sqrt(2 + ai)
```

Підкреслення тут повідомляє про те, що саме значення не є важливим для обчислень.

◀ **Приклад.** Формула *бінома Ньютона*

$$(a + b)^n = \sum_{k=0}^n C_n^k a^k b^{n-k}$$

одна з найвідоміших у математиці. Коефіцієнти $C_n^k = \frac{n!}{k!(n-k)!}$ називаються *біномними*. Обчислимо біномний коефіцієнт за заданими n і k .

Можна написати функцію для обчислення факторіалів і тричі її викликати, але це дуже нераціонально. По-перше, одні й ті самі значення, що виникають у процесі обчислення факторіала, обчислюються тричі. По-друге, функція $m!$ за збільшення m зростає дуже швидко,

тому існує чимало значень n і k , за яких значення C_n^k можна зобразити в цілих типах, а $n!$ і $k!$ — ні. (Хоча, у випадку мови Python останнє не є лімітуючим фактором.)

За умови $k \geq 1$ перетворимо формулу біноміального коефіцієнта

$$C_n^k = \frac{n!}{k!(n-k)!} = \frac{n!(n-k+1)}{k(k-1)!(n-k)!(n-k+1)} = \frac{n!}{(k-1)!(n-k+1)!} \cdot \frac{n-k+1}{k} \\ = C_n^{k-1} \cdot \frac{n-k+1}{k}$$

Маємо рекурентне співвідношення для послідовності $C_n^0, C_n^1, \dots, C_n^n$ біномних коефіцієнтів.

$$C_n^k = C_n^{k-1} \cdot \frac{n-k+1}{k}$$

Початкові умови такі: $C_n^0 = 1$.

Усі елементи послідовності цілі, тому обчислення можна задати так.

```
c = 1
for k in range(1, m + 1):
    c = c * (n - k + 1) // k
```

Зауваження. Запис $c *= (n - k + 1) // k$ є помилковим, оскільки $n - k + 1$ не обов'язково націло ділиться на k . У той же час біномний коефіцієнт є цілим числом. Тому спочатку треба виконати множення, а тільки потім ділити.

Якщо взяти до уваги тотожність $C_n^m = C_n^{n-m}$, кількість повторень циклу за $m \leq n/2$ можна зробити рівною m , інакше — $n-m$. Не забудемо також, що $C_n^m = 0$ за $m > n$ або $n < 0$. Оформимо обчислення у вигляді функції з цілими параметрами n і m , яка повертає значення типу **int**.

```
def cnm(n, m):
    """Return binomial coefficient.

    Arguments:
    n - size of total set
    m - size of combination
    pre: 0<=m<=n, else return 0
    """
    if (m > n or m < 0): return 0
    if (m > n // 2): m = n - m
    c = 1
    for k in range(1, m + 1):
        c = c * (n - k + 1) // k
    return c
```

На останок зазначимо що в модулі **math** наявна функція **comb(n, m)**, що обчислює значення біномного коефіцієнта. Проте продемонстрований метод може бути застосований і в інших схожих обчисленнях.

◀

У наведених прикладах було показано, що рекурентні співвідношення дозволяють спроектувати циклічні обчислення. Крім того вони використовуються для оптимізації обчислень (див. наступний розділ). Насправді, це такий само потужний засіб у програмуванні, що й індукція в математиці.

Під час програмування за рекурентним співвідношенням перевагу краще віддавати ітерації (циклу), а не рекурсії. У випадку рекурсивної реалізації може відбуватися експоненційна кількість викликів. Натомість обчислення за рекурентним співвідношенням у циклі найчастіше використовує кількість елементарних операцій, що лінійно залежить від кількості ітерацій циклу (що в багатьох випадках майже те саме, що й аргумент, для якого виконуються обчислення).

12.7 Переривання та продовження циклу – зло

Інструкція **break** задає переривання поточного циклу: за її виконання цикл завершується, причому дії, записані в секції **else** не виконуються; далі виконуються дії, наступні за цим циклом. Якщо **break** записано в інструкції циклу, яку вкладено в іншу інструкцію циклу, то виконання **break** завершує вкладений цикл, а зовнішній цикл продовжується.

Інструкція **continue** всередині циклу задає перехід на кінець тіла циклу. В інструкції циклу **while** слідом за **continue** обчислюється умова продовження циклу. У випадку циклу **for** інструкція **continue** задає перехід на виконання наступної ітерації (для наступного значення змінної-лічильника, якщо існує).

Використання інструкцій **break** і **continue** для циклів порушують послідовний порядок обчислення інструкцій, при цьому вони заплутують текст програми й ускладнюють її подальше тестування. Також вони ускладнюють використання інваріантів циклу для перевірки правильності побудованих циклів.

Інколи ці інструкції дійсно скорочують запис розгалужень у циклі, проте в більшості випадків цикл **while** без втрати прозорості коду *можна записати* без них.

Цикл **for** мови Python, на жаль, без них виразити можна, але у випадку **break** отриманий код скоріш за все буде виглядати дуже громіздко та неприродньо.

У довільному разі зловживати **break** та **continue** не слід. Особливо **break**.

12.8 Структурованість коду

Для структурованої програми має виконуватися правило: *кожна логічна частина коду повинна мати один вхід* (інструкцію, з якої починається її виконання) *та один вихід* (інструкцію, якою завершується її виконання). За цих умов у програмі не буде заплутаних переходів між її частинами, вона буде залишатися зрозумілою не тільки для комп'ютера, але й для програміста; також її буде легше перевіряти на відповідність специфікації та вимогам і за необхідністю виконувати наладку та модифікацію.

Інструкції **break**, **continue**, **return** здатні порушувати структурованість коду. Слід уникати грубого порушення структурованості програми. Такі ситуації виникають наприклад тоді, коли з тіла циклу за інструкцією **return** виконується вихід з функції.

Приклад. Розглянемо функцію, що повертає ознаку того, що в десятковому запису цілого числа **n** наявна цифра **7**. Перш за все параметризуємо задачу: будемо шукати не **7**, а цифру **d**. Алгоритм розв'язку буде полягати в тім, що послідовно отримувати цифри числа, поки не буде знайдено цифру **d** або не закінчатся цифри. Перше, що спадає на думку: як тільки цифру знайдено, функція може повертати результат.

```
def find_digit(n, b):
    """
    Return if digit b is present in digits of n.

    Keyword arguments:
    n - number to analyze
    b - digit to find
    pre: n>0, b>=0, else return False
    """
    if n <= 0 or b < 0: return False
    while n > 0:
        if n % 10 == b: return True    # гірше, ніж погано!
        n //= 10
    return False
```

Спробуємо позбавитися від завершення функції з тіла циклу. Будемо достроково завершувати цикл, якщо цифру знайдено. Тоді по завершенню циклу ненульове значення **n** буде ознакою того, що цифру було знайдено. Друга версія виглядатиме так.

```
def find_digit(n, b):
    """
    Return if digit b is present in digits of n.

    Keyword arguments:
    n - number to analyze
    b - digit to find
    pre: n>0, b>=0, else return False
    """
    if n <= 0 or b < 0: return False
    while n > 0:
        if n % 10 == b: break          # погано!
        n //= 10
    return n > 0
```

Ця версія краща, але все одно містить **break**, коли позбутися від переривання циклу тут доволі нескладно. Запишемо остаточну версію.

```
def find_digit(n, b):
    """
    Return if digit b is present in digits of n.

    Keyword arguments:
    n - number to analyze
    b - digit to find
    pre: n>0, b>=0, else return False
    """
    if n <= 0 or b < 0: return False
    while n > 0 and n % 10 != b:
        n //= 10
    return n > 0
```

Замість того, щоб переривати цикл, умову, за якою це відбувалося, додано до умови продовження циклу. ◀

Зауваження. У деяких випадках, щоб позбавити тіло циклу **while** від **break** варто додати допоміжну змінну з логічним значенням, що містить ознаку того, чи слід продовжувати ітерації.

У той же час, використання інструкції **return** на початку функції для обробки тривіальних або нестандартних ситуацій у багатьох випадках є припустимим й навіть доцільним, оскільки відокремлює логіку обробки помилок від виконання основних обчислювальних дій й тим самим спрощує сприйняття коду. В якості прикладу можна згадати раніше наведену функцію **fibo**.

```
def fibo(n):  
    """  
    Return the n-th Fibonacci number.  
  
    pre: n>=0, else return -1  
    """  
    if n < 0:  
        return -1  
    f2, f1 = 0, 1  
    if n == 0:  
        return f2  
    for i in range(1, n):      # f1=F_i  
        f2, f1 = f1, f1 + f2  # зсув «віконця»  
    return f1
```

Запишемо варіант цієї функції з єдиною інструкцією **return** наприкінці її тіла, використавши при цьому допоміжну змінну.

```
def fibo(n):  
    """  
    Return the n-th Fibonacci number.  
  
    pre: n>=0, else return -1  
    """  
    if n < 0:  
        res = -1  
    else:  
        f2, f1 = 0, 1  
        if n == 0:  
            res = f2  
        else:  
            for i in range(1, n):      # f1=F_i  
                f2, f1 = f1, f1 + f2  # зсув «віконця»  
            res = f1  
    return res
```

Але чи сприймається другий код краще за перший? Скоріш за все ні. (А у випадку великої кількості вкладених перевірок точно ні.) Отже, перша версія в даному випадку краща за другу, хоча й дещо порушує структурованість.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: інструкція циклу з передумовою, інструкція **while**, заголовок циклу, тіло циклу, ітерація циклу, інваріант циклу, оператор присвоювання, інструкція **for**, ітератор, лічильник циклу, **range**, рекурентне співвідношення, інструкція **break**, інструкція **continue**, структурованість коду.

2. Яке основне використання циклів?
3. Який загальний синтаксис інструкції **while**?
4. Яку семантику має інструкція **while**?
5. Яку семантику має секція **else** у випадку циклів?
6. Яку семантику має оператор присвоювання?
7. Чи наявні в мові Python цикли з післяумовою?
8. Яка інструкція може бути використана для обходу всіх елементів певної сукупності?
9. Чи можна змінювати лічильник циклу **for**?
10. Як зміни лічильника циклу **for** будуть впливати на його виконання?
11. Як у мові Python задати послідовність цілих чисел, що утворює арифметичну прогресію?
12. Чи можна за допомогою **range** задати послідовність дійсних чисел від **0.0** до **1.0** з кроком **0.1**? Чому?
13. В яких ситуаціях варто використовувати цикл **for**?
14. Чи варто використовувати цикл **for**, якщо кількість ітерацій заздалегідь невідома?
15. Який зв'язок мають цикли та рекурентні співвідношення?
16. Як можна запрограмувати обчислення за рекурентним співвідношенням?
17. Для чого використовуються рекурентні співвідношення?
18. Яке обчислення n -го числа Фібоначчі буде вимагати менше часу: за допомогою рекурсивної функції чи за допомогою функції, що використовує цикл?
19. У чому переваги структурованого коду?
20. Які інструкції здатні порушувати структурованість коду?
21. Чи варто суворо дотримуватися того, щоб код був максимально структурованим?

Задачі та вправи

1. Що буде надруковано за виконання програми?

```
i, n, m = 1, 1, 2
while n < m:
    print(i, n, m)
    i += 1
    n *= i
    m *= 2
```

а)

```
i, n, m = 1, 1, 2
while n < m:
    print(i, n, m)
    i += 1
    n *= i
    m *= 2
else:
    print(i, n, m)
```

б)

```
i, n, m = 1, 2, 1
while n < m:
    print(i, n, m)
    i += 1
    n *= i
    m *= 2
```

в)

```
i, n, m = 1, 2, 1
while n < m:
    print(i, n, m)
    i += 1
    n *= i
    m *= 2
else:
    print(i, n, m)
```

г)

2. Скільки існує різних можливих цілих значень, які можна подати на вхід програми, щоб вона вивела 768?

```
s = int(input())
s = s * 10
n = 3
while s > 0:
    s = s - n
    n = n * 2
print(n)
```

3. Скільки існує різних можливих цілих значень, які можна подати на вхід програми, щоб у результаті роботи програма вивела числа 12 та 18 (у вказаному порядку)?

```
s = int(input())
k1 = 0
```

```

k2 = 0
while s < 230:
    s += 20
    k1 += 1
while s >= 13:
    s -= 13
    k2 += 1
print(k1, k2)

```

4. Написати функцію, що за цілим $a \geq 0$ обчислює найменше n , за якого $n! \geq a$.
5. Написати функцію, яка за цілим a обчислює та повертає найбільше n , за якого $n! \leq a$.
6. Написати функцію, яка за цілими n та m обчислює та повертає $\lfloor \log_n m \rfloor$. (У передумові до функції зверніть увагу на значення функції у випадку некоректних вхідних даних.)

7. Написати функцію, яка за цілим числом визначає:

- а) кількість цифр його десяткового запису;
- б) чи зустрічається в його десятковому запису задана цифра;
- в) кількість входжень заданої цифри в його десятковий запис;
- г) старшу цифру його десяткового запису;
- д) мінімальну (максимальну) цифру його десяткового запису.

8. Що буде виведено за виконання програми?

```

n, m = 1, 6
while n < m:
    print(n, m)
    n += 1
    if n % 3 == 1:
        continue
    n += 1
else:
    print(n, m)

```

а)

```

n, m = 1, 6
while n < m:
    print(n, m)
    n += 1
    if n % 3 == 1:
        break
    n += 1
else:
    print(n, m)

```

б)

```

n, m = 1, 6
while n < m:
    print(n, m)
    n += 1
    if n % 3 == 1:
        break
    n += 1
print(n, m)

```

в)

9. Що буде виведено за виконання програми?

```

for i in range(5):
    print(i, end=' ')

```

а)

```

for i in range(1, 5):
    print(i, end=' ')
    print(i, "The end.")

```

б)

```

for i in range(1, 5):
    print(i, end=' ')
else:
    print(i, "The end.")

```

в)

```

for i in range(7):
    i = 8
    print(i, end=' ')
else:
    print(i, "The end.")

```

г)

```

for i in range(5, 5):
    print(i, end=' ')

```

д)

```

for i in range(1, 5, 2):
    print(i, end=' ')
    print(i, "The end.")

```

е)

```

for i in range(1, 5, -2):
    print(i, end=' ')
else:
    print(i, "The end.")

```

ж)

```

for i in range(5, 1, -2):
    print(i, end=' ')
else:
    print(i, "The end.")

```

з)

```

for i in range(5, -3, -2):
    print(i, end=' ')
    print(i, "The end.")

```

к)

```

for i in range(7):
    if i % 2 == 1:
        continue
    print(i, end=' ')

```

```

for i in range(7):
    if i % 5 == 3:
        break
    print(i, end=' ')

```

```

for i in range(7):
    if i % 5 == 3:
        break
    print(i, end=' ')

```

<pre>else: print(i, "The end.") л)</pre>	<pre>else: print(i, "The end.") м)</pre>	<pre>print(i, "The end.") н)</pre>
--	--	--

10. Написати функцію, яка за цілим $n \geq 1$ повертає n -й елемент послідовності, заданої рекурентним співвідношенням $a_n = 3a_{n-1} - a_{n-2}$, $n \geq 3$, $a_1 = 3$, $a_2 = -2$.

Записати версії з **while** і з **for**.

11. Написати функцію, яка за заданим цілим числом m знаходить найменше число Фібоначчі, яке більше m .

12. Написати функцію, яка за заданим цілим числом m знаходить найменший елемент послідовності, заданої рекурентним співвідношенням $a_n = 2a_{n-1} + 3a_{n-2}$, $n \geq 2$, $a_0 = 1$, $a_1 = 1$, що більше m .

13. Написати функцію з цілим параметром n , яка повертає значення a_n , обчислене за системою рекурентних співвідношень

$a_1 = 1$, $b_1 = 1$, $a_k = b_{k-1} \cdot a_{k-1} + 2 \cdot a_{k-1}$, $b_k = a_{k-1} \% b_{k-1} + 2 \cdot a_{k-1}$ за $k > 1$.

14. Написати функцію, яка за заданим n обчислює значення виразу

$$\sqrt{3 + \sqrt{6 + \dots + \sqrt{3(n-1) + \sqrt{3n}}}} \quad (n \text{ коренів}).$$

15. Написати нерекурсивну функцію, яка за заданим числом n визначає число, що утворюється оберненням його десяткового запису. Наприклад, оберненням 123 є 321, 340 — 43 (незначущі нулі відкидаються).

16. Не користуючись функціями обчислення факторіала, біномного коефіцієнта та вкладеними циклами, запрограмувати виведення послідовності

а) $\frac{C_0^0}{1}, \frac{C_2^1}{2}, \dots, \frac{C_{2n}^n}{n+1}$; б) $C_n^0, C_n^1, \dots, C_n^n$; в) $C_{n+0}^0, C_{n+1}^1, \dots, C_{n+n}^n$;

г) $C_{n+k}^k, C_{n+k+1}^{k+1}, \dots, C_{n+n}^{n+n}$ (у цій задачі допускається один раз обчислити факторіал);

д) $C_{2n}^0, C_{2n+1}^1, \dots, C_{2n+k}^k, \dots, C_{3n}^n$.

17. Написати функцію, що за заданим n обчислює значення виразу

$$\sum_{i=1}^n \frac{\sum_{j=1}^i \cos j}{\sum_{j=1}^i \sin j}$$

Тригонометричні функції для кожного аргументу мають обчислюватися тільки по одному разу.

18. Написати функцію, яка за заданими дійсними a , h та цілим m друкує значення функції $\arctg(\sin x)$ у точках $a, a+h, \dots, a+mh$.

19. Написати функцію, яка наближено знаходить максимум функції $\sin(\cos(13x)) + \cos(\ln x)$ на відрізку $[a; b]$. (Підказка: Обчислити функцію в точках відрізка з невеликим кроком h і серед отриманих значень взяти найбільше. Не забудьте про кінці відрізка.)

20. **Табулювання функції.** Нехай задано дійсні числа a, b , причому $a \leq b$ ($[a; b]$ — відрізок, на якому виконується табулювання), та додатне дійсне число h — крок табулювання. Необхідно вивести на екран два стовпчики — перший задає послідовні значення $a, a+h, a+2h, a+3h, \dots, b$ аргументу функції з відрізка $[a; b]$, а другий — значення функції $\sin x$ у відповідних точках. Незалежно від того, чи ділиться довжина відрізка націло на h , останній рядок повинен містити числа b та $\sin b$. Розв'язком має бути функція з параметрами **a, b, h**. У коментарі вказати передумови виклику функції та описати її поведінку за некоректних вхідних даних.

21. Написати функцію, яка на відрізку $[a; b]$ з кроком h табулює відразу дві функції: $\sin x$ і $\cos x$.

22. **Табулювання функції сторінками.** Написати функцію, яка на відрізку $[a; b]$ табулює з кроком h функцію $\sin x$, але після виведення кожних m рядків виводиться запит, чи продовжувати друкування. Робота завершується після відповіді «0».

23. **Метод половинного ділення.** Нехай наявна функція f мови Python, що обчислює монотонну та неперервну на відрізку $[a; b]$ дійсну функцію f , яка на кінцях відрізка приймає значення різних знаків. Знайти значення $x \in [a; b]$, для якого $f(x) \approx 0$.

24. **Алгоритм Герона.** Написати функцію, що наближено обчислює корінь квадратний дійсного числа. Для невід'ємного дійсного числа x послідовність $(y_n)_{n \in \mathbb{N}}$, визначена співвідношеннями

$$y_0 = 1$$

$$y_{n+1} = \frac{1}{2} \left(y_n + \frac{x}{y_n} \right)$$

прямує до $\sqrt{x}$. За шукане наближення прийняти той елемент y_n послідовності, для якого $|y_n - y_{n-1}| < \varepsilon$, тобто вважати, що

$$|y_n - y_{n-1}| < \varepsilon \Rightarrow \sqrt{x} \approx y_n$$

Тут ε деяке наперед задане невелике додатне дійсне число.

25. Написати функцію обчислення $\sqrt{x}$, де $x \geq 0$, за алгоритмом Герона та умовою продовження, яка відповідає нерівності $|y_n^2 - x| \geq \varepsilon$

26. Послідовність $(y_n)_{n \in \mathbb{N}}$, задана співвідношеннями

$$y_0 = (x + m - 1)/m, y_i = \frac{(m-1)y_{i-1}^m + x}{m}$$

збігається до $x^{1/m}$. Запрограмувати обчислення $x^{1/m}$ за довільного додатного дійсного x . Як потрібне число беремо перше y_n , для якого $|y_n - y_{n-1}| < \varepsilon \Rightarrow x^{1/m} \approx y_n$.

27. Обчислити значення числа π , використовуючи такий нескінченний ланцюговий дріб

$$\frac{\pi}{4} = \left[0; \frac{1}{1}, \frac{1^2}{2}, \frac{3^2}{2}, \frac{5^2}{2}, \dots, \frac{(2n-1)^2}{2}, \dots \right]$$

28. Для даного числа m знайти значення $\lfloor \sqrt{m} \rfloor$

$m = 24078045838557934240039346135161781281563716135707113170720894208196293666115$
 $321884451211550476337569720087916345818355302601724230118782144105547954476090610$
 $497353499606993862800447456828990658220691709506562030521625443281695065022884379$
 $579481609516279135534194339098697584786669213927244806587043352712334628651581457$
 $512394104134132388658445221316840742768390534673377373089039853558178357972646403$
 $444645349639358906791997967361175382362042296233547935808663987537567082577064789$
 $626864419674077125988661157027376218091730771413708477197748068844072101515191497$
 $0095703226362506437887288273484956889253939173$

29. За $n \geq 0$ значення $n!!$ («подвійний» факторіал) задається так: $0!! = 1$, $1!! = 1$, $n!! = n \cdot (n-2)!!$, якщо $n \geq 2$. Написати функцію, яка за цілим числом повертає його «подвійний» факторіал.

30. Обчислити $\sum_{k=1}^n \frac{x^k}{k!!}$, де $-1 < x < 1$, $k!! = 1 \cdot 3 \cdot \dots \cdot k$ за непарного k , $k!! = 2 \cdot 4 \cdot \dots \cdot k$ за парного k .

31. Написати функцію, яка перевіряє, чи є задане ціле число простим.

32. Написати функцію, яка для цілого числа знаходить кількість його різних простих дільників.

33. Написати функцію, яка для цілого числа знаходить суму його дільників.
34. Написати функцію, що для цілого числа знаходить найбільший натуральний степінь числа 3, на яке воно ділиться націло. Наприклад, для 18 це 2, для 6 це 1, для 11 це 0.
35. Знайти натуральне число з інтервалу від 1 до 10000 з найбільшою кількістю різних простих дільників. Якщо їх кілька, то цікавить найбільше.
36. Знайти натуральне число з інтервалу від 1 до 10000 з найбільшою кількістю різних простих дільників. Якщо їх кілька, то цікавить найменше.
37. Знайти кількість досконалих чисел, що менші заданого числа n . Число є досконалим, якщо дорівнює сумі всіх своїх додатних дільників, крім самого числа. Наприклад, 28 — досконале, оскільки $28=1+2+4+7+14$.
38. Знайти кількість п'ятизначних чисел, у десятковому запису яких є цифра b .
39. Знайти кількість шестизначних чисел, що діляться на суму своїх цифр. (Розглядаємо десятковий запис.)
40. Натуральне число є числом Армстронга, якщо сума його цифр, зведених у n -й степінь дорівнює самому числу (наприклад, $153 = 1^3 + 5^3 + 3^3$). Знайти суму чисел Армстронга, що записуються трьома або чотирма цифрами.
41. Написати нерекурсивну функцію, яка за заданим цілим числом визначає число, що утворюється оберненням його десяткового запису, при цьому парні цифри діляться на 2. Наприклад, для 123 маємо отримати 311, для 340 — 23 (незначущі нулі відкидаються).
42. Написати нерекурсивну функцію, яка за заданим цілим числом визначає число, що утворюється оберненням його десяткового запису, при цьому залишається кожен другий розряд, починаючи зі старшого. Наприклад, для 12345 маємо отримати 531, для 123456 — 531, для 340 — 3 (незначущі нулі відкидаються).
43. Написати нерекурсивну функцію, яка за заданим цілим числом визначає число, що утворюється оберненням його десяткового запису, при цьому залишається кожен другий розряд, починаючи з молодшого. Наприклад, для 12345 маємо отримати 531, для 123456 — 642, для 3240 — 2 (незначущі нулі відкидаються).
44. Знайти всі 10-значні числа, які є факторіалами. Для кожного числа виводити його та те число, для якого воно є факторіалом. Наприклад,
 $3! = 6$
45. Написати функцію, що для цілого числа знаходить найбільший факторіал, на який воно ділиться націло.
46. Написати функцію, яка знаходить ціле число з інтервалу $[a; b]$, що ділиться на найбільший степінь числа 3. Якщо таких чисел декілька, повернути найбільше. За допомогою цієї функції розв'язати задачу для 13-значних чисел.
47. Написати функцію, яка знаходить ціле число з інтервалу $[a; b]$, що ділиться на найбільший степінь числа 3. Якщо таких чисел декілька, повернути найменше. За допомогою цієї функції розв'язати задачу для 13-значних чисел.
48. Написати функцію, що знаходить найближчу пару чисел з інтервалу $[a; b]$, що мають рівно два різних простих дільники.
49. Написати функцію, що знаходить найбільший степінь числа 3, на який ділиться хоча б одне ціле число з інтервалу $[a; b]$.
50. Що буде надруковано за виконання програми? Чому?
- | | |
|---|---|
| <pre>k=1 try: while k<10: break else: print(1) except: pass else: print(2) finally: print(3)</pre> | <pre>k = 1 while k<10: try: break except: pass else: print(1) finally: print(2) else: print(3)</pre> |
|---|---|

а)

б)

51. Написати функцію, яка забезпечує введення додатного дійсного числа користувачем. Функція має допитувати користувача поки він не введе додатне дійсне.

Увага! Функція не повинна перехоплювати Ctrl-C, щоб залишити користувачу шанси припинити виконання програми, яка буде використовувати цю функцію.

52. **Консольне меню.** Написати програму, яка за введенням з клавіатури дійсним числом x виводить значення $\sin(x)$ або $\cos(x)$, поки користувачу це не набридне. Кожного разу перед введенням числа програма запитує користувача, що він зараз бажає обчислити ($\sin$ або $\cos$) чи він бажає завершити роботу. Якщо користувач бажає завершити роботу або вводить некоректні дані, то робота припиняється. Інакше у користувача запитується число та виводиться результат обчислення.

53. Модифікувати попередню програму так, щоб за некоректного введення робота не припинялася.

13 Задача наближеного обчислення суми степеневих ряду

13.1 Програмування обчислення з використанням рекурентних співвідношень

Розглянемо задачу наближеного обчислення нескінченної суми

$$S(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + \frac{(-1)^n x^{2n+1}}{(2n+1)!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Введемо позначку $a_n = \frac{(-1)^n x^{2n+1}}{(2n+1)!}$. Тоді $S(x) = \sum_{n=0}^{\infty} a_n$.

Відомо, що послідовність часткових сум (S_k) , де

$$S_k = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + \frac{(-1)^k x^{2k+1}}{(2k+1)!} = \sum_{n=0}^k a_n$$

за $-1 < x < 1$ «достатньо швидко» збігається до $S(x) = \sin x$, тобто функцію $\sin x$ на інтервалі $x \in (-1; 1)$ можна зобразити сумою степеневих ряду

$$\sin x = \sum_{n=0}^{\infty} a_n, \text{ де } a_n = \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Значення $\sin x$ за $x \in (-1; 1)$ можна обчислити наближено як перше S_n , для якого $|a_n| = |S_n - S_{n-1}| < \varepsilon$, де ε — деяка додатна константа. Це ε назовемо «точністю» обчислення (насправді, на відміну від цього конкретного прикладу, у більшості випадків точністю воно якраз і не буде).

Послідовність сум (S_k) задовольняє співвідношення

$$S_k = S_{k-1} + a_k \text{ за } k > 0, \text{ а } S_0 = a_0.$$

У першому наближенні алгоритм обчислень має такий вигляд (ім'ям **eps** позначено ε).

```
a=x; s=a; k=1;    # a = ak-1
while fabs(a)>=eps:
    обчислити a = ak
    s += a
    k += 1
```

Формула наступного члена ряду a_k проста, але, якщо застосувати її безпосередньо, то для кожного члена ряду доведеться обчислювати степінь і факторіал. Проте послідовність доданків теж неважко описати рекурентно: $a_0 = x$ і $a_k = a_{k-1} \frac{-x^2}{2k(2k+1)}$ за $k > 0$. Звідси, щоб отримати наступне значення a_k , треба лише попереднє a_{k-1} помножити на $-x^2$ і поділити на $2k(2k+1)$.

Отже, обчислювані величини описуються такою системою рекурентних співвідношень.

$$a_0 = x \text{ і } a_k = a_{k-1} \frac{-x^2}{2k(2k+1)} \text{ за } k > 0$$

$$S_0 = a_0, S_k = S_{k-1} + a_k \text{ за } k > 0.$$

За цієї системи **обчислити a = a_k**; у тілі циклу можна уточнити так.

$$a*=(-x*x)/(2*k)/(2*k+1);$$

Якщо в циклі обчислюється деякий вираз з незмінним значенням, краще присвоїти його допоміжній змінній перед циклом і використовувати її в циклі.

Покращимо наведений алгоритм. Помітимо: щоразу відбувається множення на $-x^2$, але x у циклі не змінюється! Тому можна перед циклом запам'ятати $-x^2$ в допоміжній змінній:

$x2 = -x * x$. Так само на кожній ітерації двічі обчислюється вираз $2 * k$. Уведемо поняття «подвоєний номер доданка» і в цьому значенні вживатимемо змінну **k2**, перед циклом присвоївши їй 2.

Оформимо алгоритм обчислень як функцію з параметром x ($x \leq 1$) та точністю **eps**. Імена змінних відповідають уже введеним позначенням.

```
import math
def sum(x, eps):
    """
    Обчислення функції sin(x) з точністю eps.

    pre: -1 < x < 1, eps > 0, інакше можливе зациклення
    """

    a = x
    s = a
    k2 = 2          # подвоєний номер доданка
    x2 = -x * x
    while math.fabs(a) >= eps:
        a *= x2 / k2 / (k2 + 1)
        s += a
        k2 += 2
    return s
```

Наведену функцію можна ще трохи покращити (це покращення стосується переважно компільованих мов програмування). Значення x використовується тільки один раз на початку циклу. Тому значення $-x * x$ можна зберегти не в новій змінній **x2**, а використати для цього змінну **x**, виконавши відповідне присвоювання.

У багатьох випадках при обчисленні достатньо використовувати якесь помірковано невелике значення **ε**, наприклад 10^{-6} . Таке значення для параметра **eps** функції **sum** можна вважати «стандартним». Якщо вирішити завжди в якості **eps** використовувати значення 10^{-6} й використовувати в алгоритмі цю константу, то для обчислень з кращою точністю вже прийдеться писати іншу, аналогічну, функцію. З іншого боку, якщо «стандартної» точності достатньо, то нема бажання в кожному виклику її явно вказувати (а для цього це стандартне значення треба кожного разу згадувати чи розраховувати). Отже, задамо для точності **eps** значення за умовчанням.

```
def sum(x, eps = 10e-6):
    """
    Обчислення функції sin(x) з точністю eps.

    pre: -1 < x < 1, eps > 0, інакше можливе зациклення
    """

    a = x
    s = a
    k2 = 2          # подвоєний номер доданка
    x = -x * x      # тепер x зберігає мінус квадрат вхідного значення
    while math.fabs(a) >= eps:
        a *= x / k2 / (k2 + 1)
        s += a
        k2 += 2
    return s
```

Примітка. Найбільш розповсюджена помилка під час обчислення таких сум — зсув значення **k**.

Для перевірки функції треба забезпечити, щоб в її викликах в якості x були аргументи 0, значення, близькі до 1 та -1 , а також проміжні значення, наприклад, 0.5, -0.5 ; в якості **eps** варто взяти не дуже великі додатні числа, наприклад 10^{-6} .

Додатково зауважимо: якщо значення x велике за модулем, то вказана в умові сума збігається повільно, і за великої кількості ітерацій навіть виникають числа, які не можна зобразити в типі **float**. Проте на відрізку $[-\pi; \pi]$ обчислення безпечні, а за x зовні цього відрізка слід скористатися тим, що $\sin(x+2k\pi) = \sin x$ за будь-якого цілого k .

13.2 Стратегії обробки помилок у математичних розрахунках

Розроблена функція цілком покладається на те, що її використовують з відповідними значеннями аргументів. У загальному випадку обчислення суми степеневого ряду за великих за абсолютною величиною значень x сума степеневого ряду може не збігатися. При цьому значення a_k може зменшуватися дуже повільно, але цикл завершиться і ми все одно отримаємо якесь значення, яке, зрозуміло, не може бути сумою незбіжного ряду. Або значення a_k може зростати за абсолютною величиною і в якійсь момент значенням змінної x стане нескінченність та цикл ніколи не завершиться. За нулевого або від'ємного значення **eps** цикл також не зупиниться.

Чи варто покращити безпечність функції **sum**, додавши перевірку, що параметри виклику x та **eps** відповідають попереднім вимогам на виклик функції? Якщо ми збираємося використовувати в програмі невелику кількість її викликів, то є сенс зробити так, щоб функція контролювала, чи зможе вона виконати обчислення над своїми аргументами. Але такого сорту функції частіше за все використовуються в обчислювальних алгоритмах, де викликаються сотні та тисячі разів, наприклад, для послідовних значень аргументу з деяким невеликим кроком у задачі інтегрування. Тоді має сенс забезпечити, щоб той інтервал, для значень якого викликається функція, задовольняв умовам, що накладаються на функцію. Тим самим ми виконаємо перевірку на застосовність функції **один раз** перед початком великої кількості її викликів і не будемо гальмувати чисельний алгоритм багатократними зайвими перевітками. Аналогічне стосується і перевірки значення **eps**.

14 Елементи технології розроблення програм

Деякі аспекти, що розглядаються в цьому розділі, піднімалися раніше, але не всі. Зараз зберемо все до купи.

14.1 Від задачі до готової програми

Розглянемо таку задачу: визначити периметр трикутника, утвореного трьома точками площини.

I. Розв'язання почнемо з **визначення вимог**, тобто **уточнення постановки задачі**. У формулюванні нічого не сказано про те, звідки мають взятися точки. Отже, домовимося, що точки вводить користувач. Також варто обумовити те, як мають оброблятися помилкові дані. Це **дуже погано**, але зараз знехтуємо можливістю помилок. Будемо вважати, що помилки введення (наприклад, введення qwerty замість дійсного числа) не перевіряються, а випадок виродженого трикутника вважається прийнятним.

II. Далі проведемо **аналіз задачі**. Тут треба вирішити, як будемо задавати точки (наприклад, у декартовій системі координат або полярними координатами) та за якою формулою будемо рахувати відстань. Домовимося, що точки будемо зображувати парою декартових координат, а відстань між точками (x_1, y_1) і (x_2, y_2) рахувати за формулою $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Буває так, що після проведення аналізу задачі знов повертаються до етапу уточнення постановки.

III. Тепер почнемо **проектувати** код майбутньої програми.

Спочатку запишемо **алгоритм** розв'язку в загальному вигляді та визначимо необхідні **структури даних** (у найпростішому випадку ними будуть використовувані змінні).

Кожну точку в програмі будемо зображувати двома дійсними змінними (для x- та y-координати).

Алгоритм:

- 1) ввести три точки;
- 2) обчислити периметр;
- 3) вивести результат обчислення.

Зауваження. Наведений алгоритм є типовим для багатьох проєктів, коли на верхньому рівні виділяються три логічні частини: введення даних, обчислення, виведення результатів. Якщо ці частини між собою не перетинаються, то кожную з них у подальшому можна буде налагоджувати й змінювати окремо, що **суттєво полегшить розробку** проєкту в цілому.

Наведений алгоритм має дуже абстрактні кроки, які потребують уточнення. Уточнимо його:

- 1) ввести першу точку;
ввести другу точку;
ввести третю точку;
- 2) обчислити периметр як суму попарних відстаней між введеними точками;
- 3) вивести першу, другу та третю введені точки і результат обчислення.

Спосіб проєктування, за якого потрібні дії поступово уточнюються, називають проєктуванням **згори донизу**. При роботі над великими проєктами використовується переважно він.

Спосіб проєктування, за якого спочатку деталізуються невеликі дії, з яких поступово збираються більші, називають проєктуванням **знизу догори**. Такий метод може бути корисним при моделюванні фізичного обладнання, коли треба враховувати властивості апаратної частини. Але навіть у цьому випадку знизу догори проєктується відносно невелика частина, пов'язана з фізичним обладнанням, а решта проєкту все одно виконується згори донизу.

На цьому етапі вже можна бачити окремі підзадачі:

П1 ввести точку

П2 обчислити відстань між двома точками

П3 вивести точку

Підзадачі не завжди є окремими кроками алгоритму. Якщо використання підзадач в алгоритмі не є очевидним, то алгоритм можна переформулювати в термінах підзадач.

Далі, за потреби, слід уточнити алгоритми розв'язання виділених підзадач.

Алгоритм для ПІ ввести точку:

вивести запрошення на введення точки

ввести x- та y- координати точки

У загальному випадку при уточненні алгоритмів для підзадач можуть виникнути нові, ще менші, підзадачі.

Коли всі алгоритми достатньо деталізовано, можна переходити до **кодування**, тобто написання коду.

IV. У нашому прикладі **кодування** почнемо з того, що реалізуємо підзадачі у вигляді окремих функцій.

Спочатку запишемо функцію обчислення відстані між двома точками.

```
def dist(x1, y1, x2, y2):  
    """Euclid's distance from (x1,y1) to (x2,y2)"""  
    return math.hypot(x1 - x2, y1 - y2)
```

Документація та анотація не є обов'язковими з точки зору синтаксису, але є дуже бажані з точки зору гарного стилю програмування. Вони повідомляють, що обчислює функція і які аргументи вона вміє обробляти. Узагалі, **гарно задокументовану функцію простіше використовувати**: щоб дізнатися, як це зробити правильно, достатньо тільки прочитати коротку документацію на неї; розбиратися в її коді (який може бути доволі складним) вже не потрібно.

Запишемо функцію виведення точки.

```
def writePoint(x, y):  
    print(f"({x:f}, {y:f})", end="")
```

Функція виведення нічого не повертає (поки що вважаємо, що виведення на консоль завжди вдале). Ім'я функції та імена її формальних параметрів говорять самі за себе, тому коментар тут вже буде надлишковим і замість того, щоб спрощувати програмісту розуміння коду, буде вже тільки заважати. Оскільки функція нічого не повертає, то інструкція **return** не є обов'язковою. За її відсутності виклик функції буде завершено, як тільки завершиться виконання тіла функції.

Тепер запишемо код функції введення точки.

```
def readPoint():  
    """  
    Input point.  
  
    On error raise an exception.  
    """  
    print("Enter point coordinates:")  
    x = float(input("\ntenter x: "))  
    y = float(input("\ntenter y: "))  
    return x, y
```

Якщо користувач введе нечислові дані, то спроба конвертувати їх на дійсні числа призведе до появи винятку. Цю особливість варто задокументувати, що й було зроблено.

Обробку помилок можна реалізовувати по-різному. Але в цьому випадку за відсутності введеної точки вже нема чого обробляти, тому має сенс завершувати виконання програми. Щоб не нагромаджувати умовні оператори, тут варто використати переваги винятків. Тому винятку дозволено «вийти» з функції, а далі він буде оброблятися на верхньому рівні програми.

У наведеній нижче програмі всі дії «обгорнуто» у функцію **main**, яка повертає ознаку успішності свого виконання.

```

import math
def dist(x1, y1, x2, y2):
    """Euclid's distance from (x1,y1) to (x2,y2)"""
    return math.hypot(x1 - x2, y1 - y2)

def writePoint(x, y):
    print(f"({x:f}, {y:f})", end="")

def readPoint():
    """
    Input point.

    On error raise an exception.
    """
    print("Enter point coordinates:")
    x = float(input("\ntenter x: "))
    y = float(input("\ntenter y: "))
    return x, y

def main():
    try:
        x1, y1 = readPoint()
        x2, y2 = readPoint()
        x3, y3 = readPoint()
        p = dist(x1, y1, x2, y2) + dist(x2, y2, x3, y3)\
            + dist(x1, y1, x3, y3)

        print("Your points are:")
        print(end="\t")
        writePoint(x1, y1)
        print()
        print(end="\t")
        writePoint(x2, y2)
        print()
        print(end="\t")
        writePoint(x3, y3)
        print()
        print(f"Triangle's perimeter = {p:f}")
    except Exception as e:
        print("Smth went wrong. Unable to calculate.\n", e)
        return 1
    else:
        return 0

try:
    main()
except:
    print("Smth unexpected happened.")

```

V. Після того, як код побудовано, його зазвичай перевіряють. На першому етапі перевірки слід перевірити, чи є він синтаксично правильним, та виправити всі синтаксичні помилки. Якщо код синтаксично правильний й за ним успішно створюється виконуваний файл, то далі його *тестують*, тобто перевіряють. Тестувати можна як код в цілому, так й окремі його частини. Під час тестування код виконують із заздалегідь підібраними вхідними даними та перевіряють, чи правильно вони обробляються.

Якщо при проектуванні найчастіше проєктують згори донизу, то при тестуванні — навпаки: спочатку тестують найпростіші функції, з яких будуються інші, а потім вже ті, які з них будуються, й наприкінці програму в цілому. У такий спосіб значно легше локалізувати наявні помилки (знайти місце/місця в коді, які їх спричиняють).

Якщо тестування виявило помилки, то далі можливе повторення попередніх етапів. Найчастіше етапів проєктування та кодування, але в деяких випадках можливе повернення й на більш ранні етапи.

У реальних проєктах наявні ще етапи впровадження та супроводження.

14.2 Якість програмного забезпечення

Якість програмного забезпечення можна оцінювати з різних боків. Можна, наприклад, окремо оцінювати якість проєкту та якість реалізації проєкту. Існує багато точок зору на поняття якості програмного забезпечення. Узагалі більшість критеріїв якості достатньо суб'єктивні.

Характеристики якості можна розділяти на **зовнішні** та **внутрішні**.

Зовнішні характеристики якості можуть біти оцінені користувачем і переважно стосуються інтерфейсу:

- зручність та адекватність інтерфейсу;
- можливість самостійно в ньому розібратись;
- наявність документації;
- наявність змістовної діагностики помилок;
- швидкодія інтерфейсу тощо.

Принцип найменшого здивування. У розробці інтерфейсу користувача чи мови програмування **принцип найменшого здивування** (*principle (or rule) of least astonishment (or surprise)*) стверджує те, що коли два елементи інтерфейсу конфліктують або неоднозначні, то їх поведінка повинна бути такою, щоб найменш здивувати користувача або програміста під час виникнення конфлікту, зокрема тому, що найменш непередбачувана поведінка звичайно буває правильною.

Наприклад, якщо за комбінацією клавіш **Ctrl-Q+L** повинен виконуватись якийсь макрос, а за **Ctrl-Q** закриватися програма, то за цим принципом перевага повинна бути надана виконанню макросу. (Насправді не слід так налаштовувати клавіатуру.) Також з цього принципу випливає, що в прикладній програмі повинні бути зрозумілі налаштування за умовчанням.

Внутрішні — решта, переважно стосуються коду. Розглянемо питання якості коду більш детально. Тут використовують багато параметрів.

1. **Відповідність коду специфікації та вимогам**, до якої інколи додають ще **надійність коду** (відсутність відмов програмного забезпечення у певних ситуаціях).

Найчастіше поведінка програми специфікується на достатньо високому рівні абстракції та деякі моменти залишаються поза увагою, наприклад, поведінка в нестандартних з точки зору користувача ситуаціях. Якщо коротко, то програмне забезпечення повинно працювати так, як від нього очікують (і можливо краще ☺).

2. **Безпечність** з точки зору захисту своїх даних від несанкціонованого доступу.

3. **Швидкодія**.

4. **Можливість перенесення на різні платформи**.

5. **Легкість модифікації, наладки, тестування та повторного використання** (та супроводження), сюди ж відноситься й розширюваність.

Ці характеристики суттєві як для розробника, так й для замовника, оскільки явно чи неявно впливають на споживчі властивості програмного забезпечення та/або на його вартість та вартість його обслуговування.

14.3 Фактори, що впливають на ефективність роботи з кодом

Легкість модифікації, наладки, тестування та повторного використання (та супроводження) насправді обумовлена багатьма іншими, більш конкретними характеристиками програмного забезпечення, кожна з яких спрямована на те, щоб людина могла ефективно працювати з кодом. Розглянемо їх перелік більш детально.

— **Стислість, відсутність дублювання коду та невикористовуваних фрагментів**

Чим коротший код, тим простіше його налагоджувати, тестувати, модифікувати, документувати, одним словом обслуговувати. Якщо код дублюється, то крім того, що кожен копію треба окремо налагоджувати й тестувати, то ще треба не забувати одночасно вносити модифікації в усі його копії.

— **Однорідність системи запису позначень та термінології**

Якщо для одних тих самих сутностей у різних частинах коду використовуються різні назви, то розібратись з таким кодом буде складно.

— **Структурованість**

Детально структурованість розглядалася в підрозділі 12.8.

— **Зрозумілість коду, тобто ясність його призначення**

Саме зрозумілість коду забезпечує багато інших ознак якісного коду. Багато фахівців сходяться на тому, що гарний код повинен 1) читатися як розмовна мова; 2) відокремлювати важливий код від побічного.

Мартін Фаулер (Martin Fowler) виразив це краще:

Кожний дурень може написати код, зрозумілий комп'ютеру. Гарні програмісти пишуть код, зрозумілий для людей.

У зв'язку з цим виникає низка природних запитань. А чи не можна просто до незрозумілого коду додати коментар з описом виконуваних кодом дій? Чому гарний код **повинен** бути зрозумілим людині? Справа у тому, що далі код (який у сучасних програмах вимірюється мільйонами рядків!) буде обслуговувати не комп'ютер, а людина, а саме: налагоджувати, виправляти помилки та модифікувати в разі змін вимог до нього. Саме для цього людина повинна легко розуміти код. Зрозуміло, що ніхто та ніщо не завадить додати до коду коментар. Але для ефективного обслуговування коду в подальшому цього буде замало. Отже, дійсно гарний код повинен бути зрозумілим без додаткових коментарів.

На користь цієї тези можна навести ще й такий факт. Якщо код стає дуже заплутаним та незрозумілим, то для виправлення навіть однієї помилки іноді простіше буває переписати все з чистого аркушу, ніж внести зміни в існуючий код.

Одним із засобів прояснити призначення коду та полегшити його сприйняття є **коментарі** (та інші засоби документування). Але ними не слід зловживати. Коментар біля кожної інструкції програми аніскільки не полегшить її сприйняття. Дійсно якісний код повинен містити коментарі тільки там, де вони дійсно необхідні. Коментарі, що суперечать коментованому коду, гірші за відсутність коментарів.

Зміст коментарів може бути довільним, але зазвичай у коментарях зазначають:

- що повинна виконувати програма, функція чи певний фрагмент коду;
- яку сутність моделює чи для чого використовується змінна;
- які умови повинні виконуватись **перед** виконанням певних інструкцій, щоб забезпечити їх коректність (так звані **pre-умови**, або **передумови**);
- які умови будуть виконуватись **після** виконання певних інструкцій (так звані **post-умови**, або **післяумови**);
- поведінку програми чи функції за некоректних даних.

Також у коментарях можуть фіксуватися специфічні моменти алгоритмів, наприклад, інваріант циклу або умови які мають місце по завершенні циклу.

Для уникнення зайвих коментарів використовують імена, які відображають призначення відповідних змінних та функцій. Такі імена називають **змістовними** (детальніше див. розділ 5).

14.4 Деякі ознаки поганого коду

Не завжди перша версія коду є дуже гарною. У більшості випадків код поступово поліпшується, але для цього необхідно знати, що саме слід поліпшувати. У кожному випадку все індивідуально, але є певні розповсюджені проблеми, які слід вміти розпізнавати у своєму власному коді. Оскільки найбільші проблеми виникають з кодом, який *містить повтори, незрозумілий, складний*, то саме на них спрямовані наступні ознаки поганого коду.

14.4.1 Повторюваний код

Повторюваний код (англ. *duplicated code*) — найпоширеніша й гостра проблема програмного забезпечення. У підрозділі 6.1 було наведено приклади дублювання коду та зазначено проблеми, які він породжує для програмістів. Якщо від повторюваних фрагментів коду можна позбутись шляхом введення допоміжних функцій, то чому ця проблема така поширена? Насправді ще одна проблема з повторюваним кодом полягає в тім, що наявність повторень не завжди є очевидною.

Повторення може бути як явним, так і ледь помітним. Явне повторення існує у вигляді ідентичного коду, але таке зустрічається рідше, бо такі випадки легко виявляються і дублювання позбуваються. Інколи відмінність проявляється тільки в назвах допоміжних змінних або в порядку виконання комутуючих між собою інструкцій; таке дублювання помітити вже складніше. Більш приховане повторення може проявлятися в структурах або кроках обробки (наприклад, табулювання функцій **sin** та **cos**), які на перший погляд зовсім різні, але, незважаючи на це, однакові по своїй суті.

Ще одна ситуація, коли можна казати про повторюваний код, це використання неіменованих констант, що позначають певні сутності (так звані *«магічні» константи*). Наприклад, константи 3.14 як наближення до числа π . Якщо надалі вирішимо використати більш точне наближення, наприклад 3.1415, то прийдеться відшукати й змінити всі копії цієї константи, причому **однаково**.

Щоб уникнути цієї ситуації, можна означити змінну з відповідним значенням.

Щоб позбутися магічної константи **3.14** можна використати змінну.

`MY_PI = 3.14`

Зауваження. Насправді стандартна бібліотека математичних функцій надає значення константи π . Тому в наведеному прикладі краще за все використати саме бібліотечні засоби, а наведені синтаксичні можливості мови використати в інших ситуаціях.

Приклади коду, в якому треба позбутись магічних констант.

1. У цьому прикладі необхідно отримати від користувача дійсне число з відрізка `[0; 1]`.

```
x = float(input('Input a float from [0;1]: '))
if x < 0 or x > 1:
    raise ValueError('Your number not in [0;1]')
```

Наявне дублювання тут виділено кольором. Якщо надалі межі відрізка зміняться, то код буде необхідно виправляти в кількох місцях. Отже, у цьому фрагменті необхідно ввести змінні, що позначають ліву та праву межу відрізка та крізь використовувати їх значення.

2. У наступному прикладі ми хочемо за радіусом наближено знайти довжину кола, площу кола та об'єм кулі. Для поточних потреб було вирішено, що в якості наближення до числа π можна використати число 3.

```
radius = 2.56
length = 2 * 3 * radius
square = 3 * radius * radius
volume = (4 / 3) * 3 * radius ** 3
```

Через деякий час виявилось, що потрібна більш точна оцінка, а тому треба взяти більш точне значення числа π . Які з наявних чотирьох входжень треба змінити? Зрозуміло, що автоматична заміна з цією задачею не впорається. Необхідна участь програміста, якому ще прийдеться

вникати в суть розрахунків. (Магічну константу, яка має бути замінена змінною, виділено кольором.)

14.4.2 Довга функція⁵⁰

У своєму описі цієї ознаки Фаулер і Бек наводять кілька важливих причин, через які короткі функції краще довгих. Основна причина пов'язана з розподілом логіки. Ймовірність того, що дві довгі функції містять повторюваний код, доволі велика. Як тільки їх розбити на більш дрібні, у більшості випадків можна знайти способи спільно використати їх частини.

Крім того, невеликі функції допомагають у поясненні коду. Якщо ви не розумієте, що робить деякий фрагмент коду, і виділили його в одну невелику, вдало названу функцію, тоді вам буде легше зрозуміти результуючий код. Звичайно легше розширювати й супроводжувати системи, у яких переважають невеликі функції, оскільки такі системи більш прості для розуміння й містять менше повторів.

Який кращий розмір невеликої функції? Фахівці вважають, що до десяти рядків коду, при тому, що більшість функцій використовують від одного до п'яти рядків коду. Якщо переважна більшість функцій у системі зроблено невеликими, то кілька можуть бути й побільше — звичайно, не на шкоду розумінню й за умови відсутності дублювання.

Деякі програмісти не бажають писати невеликі функції, оскільки бояться втрат швидкодії, пов'язаних із ланцюговими викликами (ланцюговими вважають кілька розташованих поспіль викликів, наприклад: `f1(); f2(); f3()`) занадто великої кількості невеликих функцій. Це хибна стратегія з кількох причин. По-перше, гарні проєктувальники передчасно код не оптимізують. По-друге, ланцюгові виклики невеликих функцій занадто мало позначаються на продуктивності. По-третє, за наявності проблем швидкодії часто можна провести реорганізацію коду для того, щоб підвищити швидкодію, не відмовляючись при цьому від невеликих функцій.

14.4.3 Складність умов (conditional complexity)

На початку роботи умовна логіка проста для розуміння й складається не більш ніж з декількох рядків коду. Але, на жаль, вона рідко залишається такою і далі. За збільшення функціональних можливостей програми умовна логіка починає ставати великою та важкою для розуміння.

Умови стають складними, коли:

- умовна логіка вибирає, який з декількох варіантів базових обчислень виконувати;
- умовна логіка контролює, який з декількох фрагментів поведінки повинен бути виконаний на додаток до основної поведінки в кожному конкретному випадку;
- складними є умовні вирази, що керують зміною станів об'єкта.

Одним із шляхів спрощення складних умовних виразів є використання в них допоміжних гарно названих функцій замість явного запису умов. Щодо логіки, яка контролює варіанти поведінки, то тут найчастіше на допомогу приходять класи, успадкування та композиція, які в більш простий та прозорий спосіб дозволяють конфігурувати виконання (але відповідні приклади далеко за межами цього посібника).

14.4.4 Непристойна демонстрація (indecent exposure)

Ця проблема виникає, коли функції, методи, класи тощо, які повинні бути невидимі для клієнтів (*клієнтом* коду називають код, який його використовує; наприклад, виклик функції можна розглядати як клієнтський код цієї функції), відкриті для них. Це означає, що клієнти знають про код, що не важливий або важливий тільки побічно. Це вносить свій внесок у складність проєкту.

⁵⁰ За Фаулером та Беком в оригіналі *метод* (long method), тобто функція в об'єктно-орієнтованому програмуванні (більш точно це функція, що є членом класу).

До непристойної демонстрації у більшості випадків можна віднести й *використання глобальних змінних*. Кожна частина програми повинна мати доступ тільки до тих змінних, які їй необхідні. Відмова від глобальних змінних дозволяє коду **краще захищати власні дані від несанкціонованих змін**. Це, у тому числі, дозволяє уникати неправильного використання змінних й **суттєво спрощує тестування**.

У деяких випадках вважається допустимим використовувати глобальні змінні для збереження загальних системних налаштувань програми (локалізація тощо) або сутностей, що мають існувати в єдиному екземплярі та впродовж усього виконання програми.

Зауваження. Мова Python не дуже сприяє приховуванню та захисту даних.

Тим не менш допоміжні функції варто відповідно іменувати (з підкреслення), щоб не звертати на них увагу клієнтів.

Контрольні запитання

1. Дати означення/стисле тлумачення понять: зовнішні характеристики якості програмного забезпечення (ПЗ), внутрішні характеристики якості ПЗ, принцип найменшого здивування, змістовні імена, надлишкові коментарі, «магічні» константи, функціональне проєктування, перевантажений обов'язками код.

2. Назвіть внутрішні характеристики якості ПЗ.
3. Які фактори впливають на ефективність роботи з кодом?
4. Чому слід уникати дублювання коду?
5. Що поганого у використанні «магічних» констант?
6. Які недоліки коду найчастіше приховують довгі функції?
7. Якими шляхами можна позбутися складних умовних виразів?

Задачі та вправи

1. Фрагмент коду забезпечує введення користувачем дійсного числа з відрізка [0,1], але в ньому наявні магічні константи. Переписати фрагмент так, щоб магічних констант у ньому не було.

```
x = float(input('Input a float from [0;1]: '))
if x < 0 or x > 1:
    raise ValueError('Your number not in [0;1]')
```

2. У наступному прикладі за радіусом наближено обчислюється довжина кола, площа кола та об'єм кулі. Для поточних потреб було вирішено, що в якості наближення до числа π можна використати число 3.

```
radius = 2.56
length = 2 * 3 * radius
square = 3 * radius * radius
volume = (4 / 3) * 3 * radius ** 3
```

Переписати фрагмент так, щоб магічних констант у ньому не було.

15 Послідовнісні контейнери

15.1 Поняття структури даних

До цього моменту було розглянуто числові типи (**int**, **float**, **bool**), значення яких зображують одиничні дані. Широкий загал задач потребує збереження сукупностей даних, загальна кількість яких наперед може бути невідома. Для збереження даних у пам'яті програми використовують різноманітні **структури даних** (англ. *data structure*). Структури даних є програмними сутностями, що дозволяють зберігати сукупності однотипних та/або логічно зв'язаних між собою даних. Найчастіше структури даних реалізуються у вигляді типу даних, що забезпечує і збереження даних, і певні операції над цими даними. Такі типи даних можуть бути вбудованими типами мови, можуть бути бібліотечними типами, а також можуть реалізовуватися власними типами програміста.

В інформатиці зустрічається поняття контейнера. Під **контейнером**⁵¹ (англ. *container*), або **колекцією** (англ. *collection*), розуміють структуру даних, що зберігає набір значень одного чи різних (але, як правило, похідних від спільного базового) типів та надає доступ до цих значень. Як правило, кількість елементів даних у контейнері може бути довільною й за відсутності явних обмежень значення елементів контейнеру можуть повторюватися. Також контейнер може дозволяти записувати та вилучати дані. Найчастіше контейнери застосовуються для зображення однотипних даних, що обробляються в однаковий спосіб.

При обробці наборів даних найчастіше використовуються операції вставки/вилучення (англ. *insert, add/delete, remove*) даних у набір, пошук елемента, перегляд елемента, перевірка чи є набір порожнім, знаходження кількості елементів, але можливі й інші. Ефективність виконання цих операцій залежить від реалізації відповідної структури даних, тобто від того, як вона влаштована всередині.

За принципами організації даних виділяють **послідовнісні** (англ. *sequence container*) та **асоціативні** (англ. *associative container*) контейнери. Розгляд почнемо з послідовнісних контейнерів, про асоціативні буде дещо пізніше.

15.2 Послідовнісні контейнери

Послідовнісні контейнери зберігають послідовності даних. Мова йде про те, що спосіб зберігання та/або доступу до елементів у контейнері задає на них лінійне впорядкування: можна казати про перший, другий, ... елемент, що зберігається в контейнері. При цьому в загальному випадку елементи можуть повторюватися. З точки зору математики можна казати, що елементи утворюють послідовність.

Історично першим послідовнісним контейнером був масив. **Масив** (англ. *array*) можна розглядати як програмну модель послідовності однотипних даних, що має фіксовану довжину. Зазвичай масив займає неперервну ділянку пам'яті, в якій послідовно розташовуються його елементи, і надає **прямий доступ** до елементів: будь-який елемент масиву доступний за його номером. Номери елементів масиву прийнято називати **індексами**.

Масиви в класичному їх розумінні в мові Python відсутні (якщо не брати до уваги бібліотеку **array**, в якій означено тип **array**, що дає можливість створювати масиви для елементів деяких типів, проте не користується такою популярністю як тип багатовимірних масивів з бібліотеки **numpy**).

⁵¹ Це цілком узгоджується з початковим уявленням про контейнер (у мові Python) як про об'єкт, що містить посилання на інші об'єкти, адже ті об'єкти відображаються у значенні, зображуваним об'єктом-контейнером.

15.2.1 Прямий та послідовний доступ до елементів

Якщо певну частину даних можна отримати безпосередньо за її номером або адресою, то кажуть, що до даних є **прямий**, або **довільний, доступ** (англ. *direct/random access*). Для послідовності це означає, що до елементів є доступ за їх номерами, тобто позиціями в послідовності. Прикладом прямого доступу також є доступ до значення змінної за її іменем (якщо нехтувати особливостями Python).

Якщо ж для того, щоб отримати значення деякого елемента послідовності (навіть маючи його номер), треба спочатку переглянути всі елементи перед ним, то це — приклад **послідовного доступу** (англ. *sequential access*). На відміну від прямого, послідовний доступ передбачає, що елементи переглядаються в заздалегідь визначеному порядку, наприклад, від першого до останнього.

Типовим прикладом структур даних з послідовним доступом до елементів є **зв'язані списки** (*linked list*), які теж відносять до послідовнісних контейнерів. Доступ до інформації, що зберігається у файлах на зовнішніх носіях, також є послідовним.

Зауваження. Не слід плутати зв'язані списки з вбудованим типом **list** мови Python.

15.3 Послідовнісні контейнери мови Python

Мова Python має низку вбудованих та бібліотечних типів, що реалізують послідовнісні контейнери (див. табл. 23).

Табл. 23 — Типи мови Python, що зображують послідовності

Назва	Коротка характеристика
list	Вбудований тип списків. Зберігає послідовності значень. Допускається зберігання значень різних типів та модифікація, як послідовності в цілому, так і зміни окремих елементів. Є змінюваним типом.
tuple	Вбудований тип кортежів. Аналогічно попередньому, але є незмінюваним типом: після створення зміни об'єкта не допускаються ⁵² .
str	Вбудований тип рядків. Зберігає послідовності символів Unicode. Є незмінюваним типом. Після створення жодні зміни чи модифікації об'єкта (та рядкового значення) не допускаються.
bytes	Вбудований тип, призначений для маніпуляцій із двійковими даними. Зберігає послідовності байтів. Є незмінюваним типом. Після створення жодні зміни чи модифікації об'єкта не допускаються.
bytearray	Вбудований тип, призначений для маніпуляцій із двійковими даними. На відміну від попереднього є змінюваним типом.
range	Цей вбудований тип також відноситься до послідовностей. Особливістю є те, що елементи послідовності не зберігаються в пам'яті, а генеруються "на льоту".
array	Означений у стандартному бібліотечному модулі array . Призначений для роботи з послідовностями однотипних елементів числових типів. Є змінюваним типом.

Крім зазначених типів варто згадати тип **ndarray** з бібліотеки **numpy** (не є стандартною бібліотекою, має встановлюватися окремо), призначений для зберігання та оброблення багатовимірних масивів (у тому числі векторів та матриць).

Примітка. Багатовимірні масиви та способи їх зображення будуть розглядатися у наступному посібнику з програмування мовою Python, що висвітлюватиме роботу з прикладними бібліотеками.

⁵² Але модифікації елементів, у залежності від їх типів, можуть бути можливі.

15.4 Найпростіші способи утворення послідовнісних контейнерів

Розгляд роботи з послідовнісними контейнерами природно почати з питання про те, як задати в програмі об'єкт відповідного типу. Розглянемо найпростіші та найнаочніші випадки.

Щодо рядків типу **str**, то раніше було розглянуто літерали, за якими утворюються об'єкти типу **str**. Літерали, що задають значення типу **bytes**, відрізняються від літералів типу **str** префіксом (**b** або **B** позначає, що це не рядок символів Unicode, а послідовність байтів; вони можуть бути «сирі», але не можуть бути «відформатованими»), а також тим, що можуть містити тільки символи з кодами від 0 до 255, (причому символи з кодами 128 та більше мають задаватися виключно через керуючі послідовності).

Для утворення списку достатньо у квадратних дужках через кому записати список його елементів. Елементи можуть мати довільний тип та задаватися виразами довільної складності (як константними, так і ні. Елементи одного того ж самого списку можуть мати різні типи.

```
n = 13
a = [1, 2, 'a', n*3]
b = [4]
c = []
```

Після виконання наведеного фрагменту коду змінна **a** буде позначати список, що зберігає послідовність з чотирьох елементів **1, 2, 'a', 39**. Змінна **b** буде позначати список з одного елемента, а змінна **c** – порожній список. Порожній список зображує порожню послідовність, тобто послідовність без елементів.

Для утворення кортежу можна в круглих дужках через кому записати список його елементів. Наприклад.

```
n = 13
at = (1, 2, 'a', n*3)
ct = ()
```

Після виконання цього коду змінна **at** буде позначати кортеж з чотирьох елементів, а змінна **ct** — порожній кортеж (тобто без елементів).

Щодо одноелементних послідовностей, то тут є нюанси. В арифметичному виразі **(4) * 1.5** вираз **(4)** сприймається як арифметичний, значенням якого є ціле **4**. Так само відбувається і з самотнім виразом **(4)**. У випадку, коли йдеться про **кортеж, що складається з одного елемента, після цього елемента слід ставити кому**. Наприклад.

```
bt = (4,)
```

Тут значенням змінної **bt** є кортеж з одного елемента. Правила синтаксису дозволяють не ставити круглі дужки навколо елементів, а також залишати в кінці «зайву» кому (і для списків також), наприклад.

```
at1 = 1, 2, 'a', n*3
at2 = 1, 2, 'a', n*3,
bt1 = 4,
```

Варто усвідомлювати, що об'єкти списків та кортежів не зберігають елементи послідовності безпосередньо. Натомість вони зберігають ідентифікатори (identity) об'єктів, що є значеннями елементів. Інакше кажучи, вони зберігають не самі елементи, а тільки посилання на них.

Згідно принципу абстракції даних документація мови Python не фіксує принципи зображення списків, кортежів та рядків у пам'яті. Тому наступні рисунки відображають стан речей дуже приблизно.

Нехай утворено список.

```
s = [13, 4, 3, 1]
```

Під час аналізу алгоритму його можна спрощено уявляти так, як зображено на рис. 35: розташовані поспіль елементи. У випадку числового списку це, однозначно, зручно. Проте, за більш складних випадків таке спрощення може бути причиною помилок.



Рис. 35 — Зовсім схематичне зображення списку в пам'яті, що не має майже нічого спільного з реальністю

Елементи, з яких складено список, є окремими об'єктами. Період їх існування в загальному випадку не обмежується періодом існування об'єкта списку. Ці об'єкти в жодному разі не є частиною списку. Об'єкт списку зберігає не самі об'єкти своїх елементів, а посилання на них.

Якщо прагнути більш реалістичної картини, то список можна уявляти так, як це зображено на рис. 36 а. Таке зображення доцільно використовувати під час аналізу алгоритму чи коду, коли елементами списку (кортежу тощо) можуть бути не тільки числові значення, але й якісь інші контейнери, наприклад, списки. Чи відповідає воно дійсності? Для списків — абсолютно ні. Бо під час роботи програми довжина списку може збільшуватись. Але об'єкт списку має залишатись тим самим. Більш наближена до дійсності уява зображена на рис. 36 б: об'єкт списку зберігає посилання на допоміжний об'єкт, який зберігає посилання на елементи. Але й вона також не є точною. Але це вже технічні подробиці реалізації, які для успішного використання списків знати зовсім не потрібно.

Надалі для зображення суто числових списків може використовуватись зовсім спрощене зображення з рис. 35. У решті випадків буде використовуватись спрощене зображення, аналогічне наведеному на рис. 36 а.

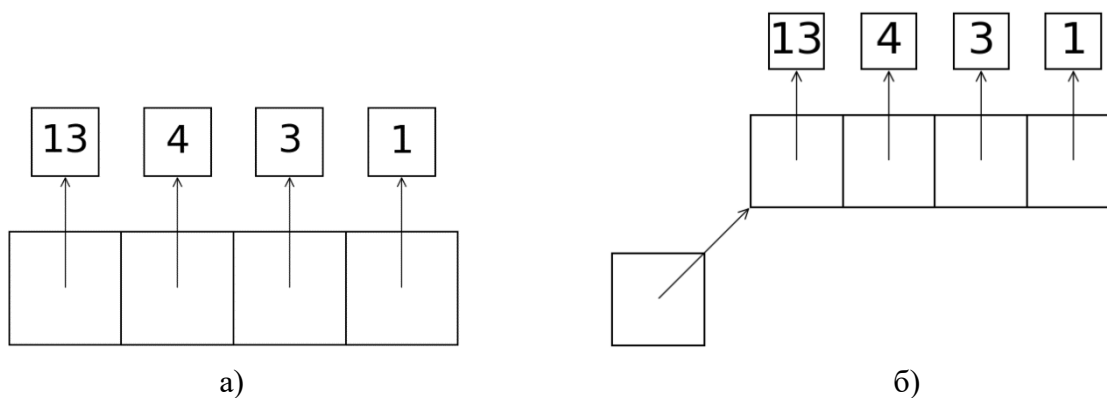


Рис. 36 — Схематичні зображення списку в пам'яті

Елементи списків та кортежів можуть мати довільні типи, у тому числі бути списками та кортежами. Але ці, більш складні, випадки розглянемо пізніше. На поточний момент обмежимося «одновимірним» випадком, коли елементи послідовності є «одинокими» елементами.

15.5 Доступ до елементів

15.5.1 Довжина та істиннісne значення послідовності

Для контейнерів типовою операцією є визначення кількості елементів, що зберігаються в контейнері, або розміру (англ. *size*) контейнеру. У випадку послідовності розглядають **довжину послідовності**. Довжину послідовності значень можна визначити за допомогою вбудованої функції **len**. На продовження попереднього прикладу, значеннями виразів **len(a)**, **len(at)**, **len(b)**, **len(ct)** є цілі 4, 4, 1, 0 відповідно. Значеннями виразів **len('Україна')**, **len('12\t12')** є цілі 7 та 5 відповідно. Довжина рядка — це довжина послідовності символів Unicode, що утворюють рядкове значення, а не довжина кількості символів, що зображують літерал, і не кількість байтів, які ці символи займають в пам'яті (нагадаємо: у форматі UTF-8, який використовується в Python за умовчанням, кириличні символи кодуються кількома байтами).

Під час перетворення на тип **bool** та оцінювання істинності непорожні послідовності вважаються істинними, а порожні — хибними. Для послідовності **s** умова **if s:** фактично означає таке: якщо послідовність **s** непорожня.

15.5.2 Індексція елементів послідовності

Вбудовані послідовнісні контейнери мови Python підтримують операцію прямого доступу, тобто *доступу за індексом*. Індексція елементів починається з нуля. Якщо в контейнері **seq** зберігається **n** елементів, то його елементи індексуються значеннями **0, 1, ..., n-1**.

Так, вираз **seq[0]** позначає елемент з індексом **0** (перший елемент послідовності), **seq[3]** — елемент з індексом **3**, **seq[n-1]** — елемент з індексом **n-1**. Спроба звернутися до елементів з індексами **n** та більше (їх у послідовності не існує!) призведе до генерації винятку типу **IndexError**.

Специфікою мови Python є можливість використання від'ємних індексів. Семантика від'ємних індексів полягає в тім, що від'ємний індекс задає індексацію не з початку, а з кінця послідовності. Так, вираз **seq[-3]** задає третій елемент з кінця послідовності, **seq[-1]** — останній елемент послідовності (перший з кінця), **seq[-n]** — перший елемент послідовності (останній, якщо йти з кінця).

Спроба звернутися за індексом, що менше **-n**, призведе до генерації винятку типу **IndexError**. Отже, для послідовнісного контейнера з **n** елементами допустимими індексами є **-n, -n+1, ..., -1, 0, 1, ..., n-1**. Індеси **-k** та **n-k**, де **k** від **1** до **n**, задають один той самий елемент.

На рис. 37 зверху та знизу відповідних елементів списку **[13, 4, 3, 1]** наведено їх можливі індекси.

0	1	2	3
13	4	3	1
-4	-3	-2	-1

Рис. 37 — Список з чотирьох елементів та їх можливі індекси

15.5.3 Обходи послідовностей

Щоб обійти послідовність **s**, можна обійти всі її елементи з індексами від **0** до **len(s)-1**, наприклад.

```
s = [10, 11, 12]
for i in range(len(s)):
    print(s[i], end=';')
```

Стандартні послідовнісні бібліотечні контейнери мови Python, про які зараз йде мова, допускають ітерування по собі. З цього випливає, що їх елементи можна обходити за допомогою циклу **for**. Наприклад, можливий такий код

```
for elem in s:
    print(elem, end=';')
```

за виконання якого, так само, як і в попередньому прикладі, буде виведено

```
10;11;12;
```

За виконання **for i in 'qwerty': print(i, end=';')** буде виведено **q;w;e;r;t;y;**. Додамо, що окремі символи рядка за межами рядка зображуються як рядки, що складаються з одного символу.

15.6 Застосовні до послідовностей оператори

15.6.1 Порівняння послідовностей

Однотипні рядки (**str**, **bytes**, **bytearray**) можна порівнювати за допомогою **==**, **!=**, **<**, **<=**, **>**, **>=**. Порівняння відбувається лексикографічно.

Щодо послідовностей інших типів, то вони вважаються рівними, якщо є послідовностями одного типу та рівні поелементно. Кортежі та списки можна порівнювати і за допомогою операторів **<**, **<=**, **>**, **>=**. У цьому випадку порівнюватись мають однотипні послідовності, порівняння відбувається лексикографічно.

15.6.2 Конкатенація та повторення послідовностей

У математиці результатом **конкатенації послідовностей** **a1, a2, ..., an** та **b1, b2, ..., bm** є послідовність **a1, a2, ..., an, b1, b2, ..., bm**, що є результатом дописування другої послідовності в кінець першої. Відповідна операція над послідовнісними контейнерами підтримується мовою Python і записується за допомогою оператора **+**. Для контейнерів незмінюваних типів її результатом завжди є новий об'єкт. (Узагалі, утворення нового значення цілком логічно й для змінюваних типів, бо для дописування другої послідовності в кінець першої зазвичай є оператор **+=**.) Для змінюваних типів документація відсилає до документації на конкретний тип. Щодо типу **list**, то чи дійсно буде утворено новий об'єкт офіційна документація замовчує. Емпірична перевірка свідчить, що таки новий об'єкт списку за обчислення **+** утворюється. Конкатенація послідовностей різних типів призводить до генерації винятку типу **TypeError**.

До послідовностей застосовна операція **n**-кратного **повторення** (англ. *repetition*) послідовності. Так, для послідовності **s** вирази **s * 2** та **s + s** задають одну ту саму послідовність значень (і того самого типу). Вираз **s * n**, де **n** — натуральне додатне число, задає послідовність, що складається з **n** послідовних копій послідовності **s**. Якщо **n** — нуль або від'ємне, то вираз **s * n** задає порожню послідовність. Вираз **n * s** позначає те саме, що й **s * n**. Наприклад, для **s = [1, 2]** вираз **s * 3** задає список **[1, 2, 1, 2, 1, 2]**. Вирази **(1, 2, 3) * 0** та **(1, 2, 3) * -2** та задають те саме, що й **()** (порожній кортеж).

До типу **range** оператори **+** та ***** незастосовні.

Зауваження. Конкатенація та повторення мають певні нюанси: під час їх обчислення над списками чи кортежами нові об'єкти, що зберігають окремі елементи, не створюються. Інакше кажучи, за виконання

```
s = [1.0, 2.0]
s2 = s*2
print(id(s2[0])==id(s2[2])==id(s[0]))
```

буде виведено

```
True
```

бо окремі елементи нового утвореного списку зв'язуються з об'єктами, з якими були зв'язані елементи початкового списку.

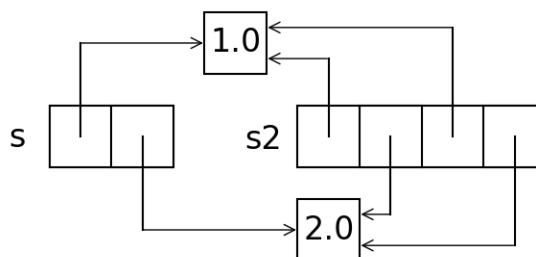


Рис. 38 — Список та результат його повторення

За послідовностей, що складаються зі змінюваних елементів, наприклад, «багатовимірних» послідовностей (будуть розглядатися пізніше), цей ефект може виявитися шкідливим (див. рис. 38).

15.6.3 Вхідження елементів у послідовність

Перевірка входження елемента в послідовність може бути виконана за допомогою операторів **in** та **not in**. Якщо **s** є списком або кортежем, то вираз **elem in s** приймає значення **True** тоді і тільки тоді, коли елемент, що дорівнює значенню виразу **elem**, входить в послідовність **s**; інакше значенням виразу є **False**. Оператор **not in** повертає значення, протилежне до результату перевірки **in**. Для текстових послідовностей (**str**, **bytes**, **bytearray**) оператори **in** та **not in** задають перевірку входження підпослідовності в послідовність символів.

Кількість входжень елемента **elem** у послідовність **s** може бути знайдена як значення виразу **s.count(elem)**. Якщо цікавить саме входить чи ні, то слід використовувати оператори **in** та **not in**, а не рахувати кількість входжень.

Зауваження з синтаксису. Типи даних, що є класами, об'єднують у собі дані та застосовні до них операції. Дані, операції над ними та інше, що об'єднує в собі клас, прийнято називати **членами** (англ. *member*) класу, а самі операції — **методами** (англ. *method*). Методи по суті є функціями, оформленими в спеціальний спосіб. Члени класів, що зберігають дані, можуть називати **атрибутами**, або **полями** (по аналогії зі структурами). Виходячи зі специфіки мови, у Python термін «атрибут» інколи застосовують у більш широкому розумінні: як синонім терміну «член класу».

Значення типу даних, що є класом, прийнято називати **екземпляром** (англ. *instance*) класу. У деяких мовах програмування замість «екземпляр» уживають термін «об'єкт». Інколи під об'єктом розуміють екземпляр, розташований у пам'яті. (І на сьогодні цей погляд є доволі розповсюдженим.) В інтернеті можна знайти багато дискусій на тему як більш правильно: екземпляр чи об'єкт. З точки зору чистої теорії ці слова синоніми. З точки зору конкретної мови програмування вони можуть позначати різні поняття. У контексті мови Python об'єкт є одиницею зображення даних у пам'яті програми. У тому числі значення-екземпляри зберігаються в об'єктах.

Отже, функції, що означені в класі, є методами, що під час виконання застосовуються до об'єктів класу. Якщо вираз **obj** позначає об'єкт класу **T**, вираз **метод** позначає метод, означений для класу **T**, то вираз **obj.метод(перелік аргументів)** позначає застосування методу з відповідними аргументами до об'єкта, тобто виклик функції.

Метод (тобто функція) **count** насправді є в кожному класі вбудованих послідовнісних контейнерів. Отже вираз **s.count(elem)** позначає виклик для об'єкта **s** методу **count**, що підраховує кількість входжень елемента **elem** у послідовність, зображувану об'єктом **s**.

15.7 Копіювання послідовностей

Послідовності можна копіювати, тобто створювати новий об'єкт, що зберігає копію послідовності.

15.7.1 Глибоке та неглибоке копіювання

Якщо змінна **s** позначає список, то за виконання виклику **s.copy()** буде утворено новий об'єкт списку, що зберігає ту саму послідовність елементів, що й список **s**. Так, за виконання інструкції

```
s2 = s.copy()
```

буде створено змінну **s2**, що зберігатиме копію списку **s**. При цьому відбудеться **неглибоке копіювання** (англ. *shallow copy*).

Неглибоке копіювання полягає в тім, що об'єкти, що є складовими частинами контейнера, не копіюються, натомість виконується копіювання їх ідентифікаторів. У випадку послідовностей це означає, що утворюється новий об'єкт, що зберігає послідовність, але його елементи — ті самі

об'єкти, що є об'єктами оригіналу. Як наслідок, оригінал та копія спільно посилаються на одні ті самі значення (див. рис. 39).

Якщо б під час копіювання утворювалися копії елементів (і, рекурсивно, це відбувалося б на кожному рівні вкладеності), то мова йшла б про глибоке копіювання (див. рис. 39). У результаті **глибокого копіювання** створюється повна копія даних і оригінал та копія не мають між собою жодних спільних частин (за винятком, можливо, окремих незмінюваних об'єктів, які зберігають значення, що зазвичай існують в пам'яті програми в єдиному екземплярі, наприклад, значення **None**, **True**, **False**, деякі цілі числа тощо).

Глибоке копіювання може мати сенс тільки в тому випадку, коли елементи послідовності є змінюваними. Щоб зробити глибоку копію, слід імпортувати модуль **copy** та використати функцію **deepcopy**. Приклад утворення глибокої копії списку наведено нижче.

```
import copy
s3 = copy.deepcopy(s)
```

Глибоке копіювання застосовне не завжди, бо в деяких випадках глибокі копії елементів створені бути не можуть (це визначається типами елементів).

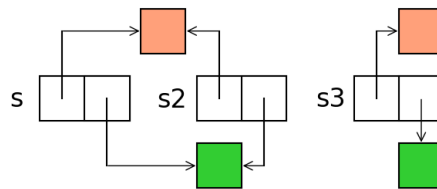


Рис. 39 — Список (ліворуч) та його неглибока (по центру) та глибока (праворуч) копії

Функція **copy** з модуля **copy** створює неглибоку копію. Серед інших типів, вона застосовна то кортежів та рядків (в яких методу **copy** на відміну від списків нема). Також для неглибокого копіювання послідовностей можна використовувати зрізи, про які буде далі.

15.7.2 Присвоювання та копіювання списків

Згадаємо, що в результаті виконання присвоювання виконується зв'язування імені з об'єктом. Розглянемо, на що це впливає у випадку послідовностей. За виконання програми

```
a = [[1, 2], [3, 4]]
b = a
```

імена **a** та **b** зв'язано з одним тим самим об'єктом списку (див. рис. 40).

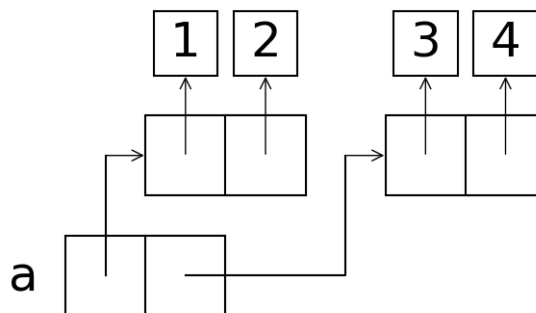


Рис. 40 — Імена **a** та **b** зв'язані з одним тим самим об'єктом

Якщо необхідно утворити копію списку, можна написати, наприклад, так.

```
c = a.copy()
```

Тепер буде утворено **неглибоку** копію списку (див. рис. 41), тому за виконання наступної інструкції, значення зображуваних списками **a** та **c** даних будуть змінюватись одночасно, хоча імена **a** та **c** позначають різні об'єкти.

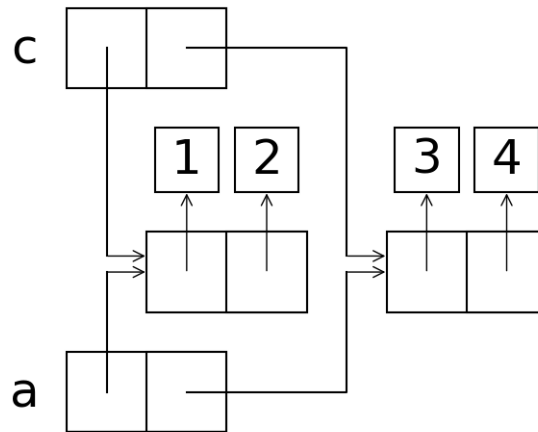


Рис. 41 — Список *c* є неглибокою копією списку *a*

`c[1][1] = 5`

Значення елементів списків *a* та *c* завдяки неглибокому копіюванню змінюються синхронно (див. рис. 42).

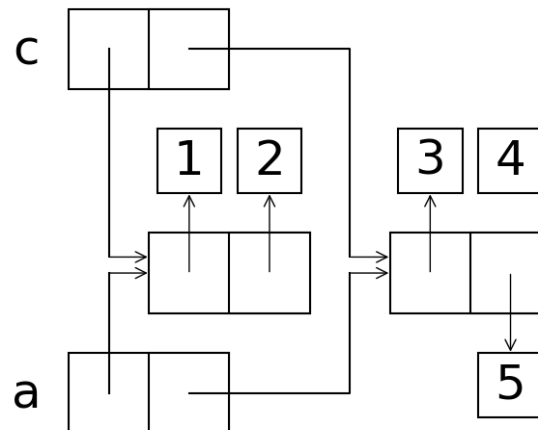


Рис. 42 — Значення зображувані списками *a* та *c* змінюються синхронно

`c[0] = 6`

Стан пам'яті після виконання останньої інструкції наведений на рис. 43).

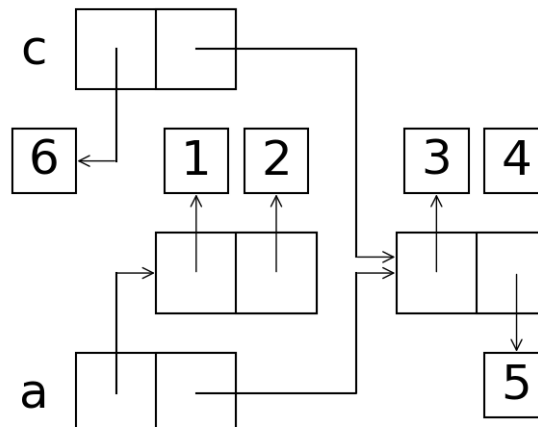


Рис. 43 — Результуючий стан пам'яті

Списки, елементами яких є списки, інколи дуже неформально, по аналогії з багатовимірними масивами, називають «багатовимірними» списками. Приблизно те саме може відбуватися і з іншими послідовнісними контейнерами.

Наведений у прикладі список **a** є «двовимірним» у тому розумінні, що його елементами є «одновимірні» списки, що складаються з одиноких елементів. Двовимірні списки можуть зображувати, наприклад, матриці. З наведеного прикладу стає зрозумілим, що у випадку «багатовимірних» структур виконувати копіювання слід дуже уважно, щоб раптом за неувважності не створити неглибоку копію там, де необхідно виконати глибоке копіювання. Зокрема, під час обчислень копіювання числових матриць, зображуваних списками чи кортежами, скоріш за все має бути глибоким (хоча нюанси залежать від алгоритму, який реалізується).

15.7.3 Утворення послідовнісних контейнерів за допомогою явного виклику конструктора

Життєвий цикл екземпляра класу починається з того, що він отримує якесь початкове значення, також кажуть, що він *конструюється*. Для цього використовується спеціальний метод класу, який називають *конструктором*. Наприклад, вираз `int('123')` задає створення об'єкта класу `int` — значення типу `int`, — і встановлює в якості його початкового значення ціле число, запис якого міститься в рядку-аргументі. Насправді тут викликається конструктор⁵³ класу `int`, і цей конструктор вміє перетворити рядок на ціле. Узагалі, встановлення початкового значення під час створення називають *ініціалізацією*.

Вираз вигляду *ім'я класу(перелік аргументів)* задає конструювання об'єкта відповідного класу за наданим набором аргументів.

Якщо порівняти вирази `123` та `int('123')`, то в першому випадку конструктор цілого числа викликається неявно, а в другому — явно. У попередніх випадках за виразами вигляду `[1, 2]`, `(1,)` конструктори послідовностей відповідних типів викликалися неявно. У той же час утворити нові об'єкти послідовнісних контейнерів можна і за допомогою явного виклику конструктора.

Якщо вираз `source` позначає об'єкт, що допускає ітерування, то вираз `list(source)` задає утворення списку, що складається з елементів із `source`; аналогічно можна утворити кортеж `tuple(source)` з тих самих елементів. Копії елементів не створюються.

Наприклад, за виконання

```
a = list('qwerty')
print(a)
b = tuple('qwerty')
print(b)
```

буде утворено список та кортеж з символів рядка `'qwerty'` та виведено.

```
['q', 'w', 'e', 'r', 't', 'y']
('q', 'w', 'e', 'r', 't', 'y')
```

Наведений спосіб дозволяє копіювати послідовності, а також перетворювати послідовності одного типу на послідовності іншого типу, зокрема, рядки на списки та кортежі, а також перетворювати списки та кортежі між собою. При цьому відбувається неглибоке копіювання.

Зауваження. Для списку `s` вираз `list(s)` задає утворення нового списку, що є неглибокою копією списку `s`.

15.8 Відрізки послідовностей

У багатьох операціях з послідовностями виникають відрізки послідовностей, над якими слід виконати дії, наприклад, для послідовності з 15 елементів можна розглядати її відрізок з 3-го по

⁵³ Зараз обмежимося згадуванням просто конструктору; хоча в Python у зв'язці працюють два методи: один розташовує об'єкт у пам'яті, інший встановлює початкове значення.

9-ий елементи. У зв'язку з тим, що допускаються від'ємні індекси, варто уточнити, як в мові Python розуміються діапазони індексів, коли йдеться про відрізки послідовностей.

15.8.1 Обчислення відрізків послідовностей

Доволі багато операцій над послідовностями потребують задання відрізка послідовності, або діапазону її елементів, над яким має виконуватися дія.

Як це прийнято в мовах програмування, при зазначенні діапазону вказується індекс **from** елемента послідовності, з якого діапазон починається, та індекс елемента **to**, перед яким діапазон завершується (елемент з індексом **to** в діапазон не входить). Оскільки мова допускає від'ємні індекси, то більш точно (але все одно потребує уточнення) буде сказати, що діапазон [**from**, **to**) складається з частини послідовності, початок якої має індекс **from**, а кінець — **to-1** (див. рис. 43)



Рис. 44 — Відрізок послідовності [**from**, **to**)

Сформулюємо правила визначення відрізка послідовності за індексами **from** та **to**. Нехай послідовність складається з **n** елементів.

1. Якщо значення **from** від'ємне, воно замінюється на **from+n** (для елемента з індексом **from** знаходимо його невід'ємний індекс).

2. Якщо значення **to** від'ємне, воно замінюється на **to+n** (для елемента з індексом **to** знаходимо його невід'ємний індекс).

3. Якщо значення **from** й досі від'ємне, воно замінюється на **0** (обрізаємо лівий кінець діапазону по початку послідовності).

4. Якщо значення **to** більше **n**, воно замінюється на **n** (обрізаємо правий кінець діапазону по кінцю послідовності).

5. Якщо після виконаних перетворень **from** $\geq$ **to**, то шуканий відрізок вважається порожнім. У решті випадків шуканий відрізок послідовності складається з елементів з індексами **from**, **from+1**, ..., **to-1** (використовуються перетворені значення **from**, **to**).

Приклад. Нехай послідовність складається з **10** елементів. Приклади обчислення відрізків наведені в табл. 24.

Табл. 24 — Обчислення відрізків послідовностей

from	to	Індекси елементів послідовності, що входять у її діапазон [from , to)
4	7	4, 5, 6
4	11	4, 5, 6, 7, 8, 9
5	2	діапазон порожній
4	-1	4, 5, 6, 7, 8
-4	9	6, 7, 8
-4	-2	6, 7
-20	45	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
-2	-5	діапазон порожній

4

15.8.2 Пошук входження елемента

Для пошуку входження елемента в послідовність можна використати метод **index**, який повертає індекс першого входження елемента в послідовність; якщо елемент не знайдено, то генерується виняток типу **ValueError**. Метод **index** дозволяє керувати областю пошуку, зокрема задавати діапазон елементів послідовності, серед яких слід виконувати пошук. За виклику **s.index(x, from, to)** пошук відбуватиметься у діапазоні [**from**, **to**) (правила його обчислення

наведені в підрозділі 15.8.1). Діапазон пошуку можна не вказувати або вказувати частково. Якщо третій аргумент (параметр **to**) не вказано, то він вважається рівним довжині послідовності і пошук відбувається до кінця послідовності. Якщо жоден з індексів не вказано, то пошук відбувається у всій послідовності.

Приклад. Нехай виконано `s = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]` та означено змінну `x`. Приклади діапазонів пошуку наведені в табл. 25.

Табл. 25 — Діапазони пошуку

Виклик	Частина послідовності, в якій виконується пошук	Примітка
<code>s.index(x, 4, 7)</code>	14, 15, 16	
<code>s.index(x, 4, 9)</code>	14, 15, 16, 17, 18	
<code>s.index(x, 4)</code>	14, 15, 16, 17, 18, 19	Останній індекс не задано: пошук відбувається до кінця послідовності.
<code>s.index(x)</code>	10, 11, 12, 13, 14, 15, 16, 17, 18, 19	Жоден з індексів не задано: пошук відбувається у всій послідовності.
<code>s.index(x, 5, 2)</code>		Елемент з індексом from розташовано після елемента з індексом to , отже задано порожній діапазон пошуку. Елемент знайдено не буде, тому буде згенеровано виняток типу ValueError .
<code>s.index(x, 4, -1)</code>	14, 15, 16, 17, 18	Індекси -1 та 9 задають один той самий елемент послідовності, тому пошук відбувається у частині послідовності з індексами від 4 до 8 .
<code>s.index(x, -5, -2)</code>	15, 16, 17	
<code>s.index(x, -2, -5)</code>		Елемент з індексом -2 розташований після елемента з індексом -5 . Діапазон пошуку порожній. Буде згенеровано виняток типу ValueError .

◀

15.8.3 Операція зрізу

Однією з важливих операцій мови Python, що застосовна до послідовностей, є операція *зрізу* (*slice*). Для вбудованих типів результатом зрізу є **копія** частини оригінальної послідовності, що утворюється як результат неглибокого копіювання. Результат зрізу має той самий тип, що і послідовність, за якою він утворений: для списку результатом зрізу є список, для кортежу — кортеж, для рядка — рядок.

Нехай змінна `s` позначає послідовність. Вираз `s[from:to]` задає копію (неглибоку) частини послідовності від елемента з індексом **from** (включно) до елемента з індексом **to** (не включаючи), тобто відрізка послідовності **[from, to)** (правила його обчислення наведені в підрозділі 15.8.1). Згідно правил обчислення відрізка, якщо значення **from** менше допустимих індексів елементів послідовності, то початком зрізу вважають елемент з індексом **0**. Якщо **to** перевищує `len(s)`, то воно замінюється на `len(s)`. Порівняйте: в аналогічній ситуації операція доступу за індексом генерує виняток.

Нехай виконано присвоювання `s=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. Тоді зрізи `s[0:2]`, `s[-3:8]`, `s[-5:-8]`, `s[-22:15]`, `s[20:-15]` задають списки `[0, 1]`, `[7]`, `[]`, `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`, `[]` відповідно.

Операція зрізу дозволяє не вказувати один або обидва індекси. Якщо перший індекс опущено або він дорівнює **None**, то він за умовчанням розуміється як **0**. Якщо опущено другий індекс (або

він дорівнює **None**), то він за умовчанням розуміється як **len(s)**. Так, **s[:5]** задає ту саму послідовність, що й **s[0:5]**, **s[1:]** відповідає **s[1:len(s)]**, а **s[:]** — **s[0:len(s)]**.

Невелике, але корисне зауваження. Щоб утворити копію послідовності **s**, можна використати вираз **s[:]**.

По аналогії з **range** дозволяється робити вибірку не всіх елементів із заданого діапазону, а з певним кроком; відповідний вираз має вигляд **s[from:to:step]**, де параметр **step** задає крок. За додатного **step** з відрізка послідовності **[from, to)** з кроком **step** вибираються елементи. Так, вирази **s[:3]**, **s[-5:10:2]** задають списки **[0, 3, 6, 9]** та **[5, 7, 9]** відповідно.

Крок не може дорівнювати нулю, але може бути від'ємним. У цьому випадку виконуються дії, аналогічні обчисленню відрізка **[from, to)** у загальному випадку:

1. Якщо значення **from** від'ємне, воно замінюється на **from + len(s)** (для елемента з індексом **from** знаходимо його невід'ємний індекс).

2. Якщо значення **to** від'ємне, воно замінюється на **to + len(s)** (для елемента з індексом **to** знаходимо його невід'ємний індекс).

3. Якщо значення **to** й досі від'ємне, воно замінюється на **-1** (обрізаємо по кінцю послідовності).

4. Якщо значення **from** більше дорівнює **len(s)**, воно замінюється на **len(s)-1** (обрізаємо по кінцю послідовності).

Якщо після виконаних перетворень **from ≤ to**, то шуканий зріз **s[from:to:step]** порожній; інакше з елементів послідовності з індексами **from**, **from+step**, ... **from+k*step**, де **from+(k+1)*step ≤ to < from+k*step** утворюється послідовність, що є значенням зрізу **s[from:to:step]**.

У випадку від'ємного кроку відсутні індекси **from** та **to** інтерпретуються як **len(s)-1** та **-len(s)-1**. Так, вираз **s[::-1]** задає список **[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]**. Вираз **s[4::-1]** задає список **[9, 8, 7, 6, 5]**, а **s[4:-1:-1]** задає порожній список (бо позиція **-1** відповідає останньому елементу списку, а не позиції перед початком).

15.9 Операції, що застосовні до послідовностей змінюваних типів

Для послідовностей змінюваних типів можливі операції, які змінюють послідовність у цілому (наприклад, додають чи вилучають з неї елементи), та/або окремі елементи послідовності. Тип **list** є змінюваним, тому надалі будемо вважати, що значенням змінної **s** є список.

Списки дозволяють зміну значень елементів, а також заміну та вставку елементів у довільній позиції списку. Розглянемо більш детально.

Перш за все, елементи списку можна змінювати. Тобто вони можуть бути «цільовими» змінними в присвоюванні, наприклад **s[i] = x**. За останньої інструкції елемент списку **s** з індексом **i** зв'язується з об'єктом, що є значенням виразу **x**.

Приклад.

```
s = [13, 4, 3, 1]
s[2] = s[3]
```

Стан пам'яті до і після виконання останнього присвоювання наведений на рис. 45.

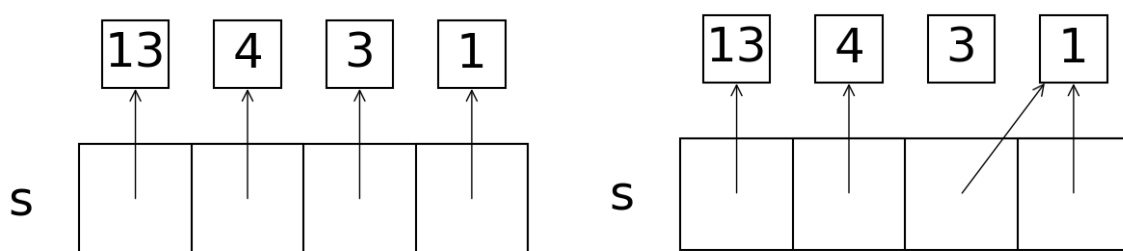


Рис. 45 — Список **s** до та після присвоювання елементу

◀

15.9.1 Вставка та вилучення елементів

Якщо вираз **t** позначає об'єкт, по якому можна ітерувати, то за присвоювання **s[i:j] = t** відповідний фрагмент послідовності **s** замінюється на елементи, що беруться з **t**. Якщо більш точно, то елементи з **t** додаються перед позицією **i**, а далі вилучаються елементи від позиції **i** до **j-1** включно.

Присвоювання **s[i:j:k] = t** можливе тільки у випадку, коли кількість елементів зрізу **s[i:j:k]** у точності дорівнює кількості елементів **t**: відповідні елементи списку **s** замінюються на елементи з **t**.

Приклад. У табл. 26 показано результат виконання операцій над списками.

Табл. 26 — Покрокове виконання операцій над списками

Програма	Вихід програми
<code>s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</code> <code>s[2:6] = ('a', 'n')</code>	
<code>print(s)</code>	<code>[0, 1, 'a', 'n', 6, 7, 8, 9]</code>
<code>s[4:1]=(13,15)</code>	
<code>print(s)</code>	<code>[0, 1, 'a', 'n', 13, 15, 6, 7, 8, 9]</code>
<code>s[40:40]=(2,2)</code>	
<code>print(s)</code>	<code>[0, 1, 'a', 'n', 13, 15, 6, 7, 8, 9, 2, 2]</code>
<code>s[1:12:3]=(-3,-3,-3,-3)</code>	
<code>print(s)</code>	<code>[0, -3, 'a', 'n', -3, 15, 6, -3, 8, 9, -3, 2]</code>
<code>s[6::-4]=(11,11)</code>	
<code>print(s)</code>	<code>[0, -3, 11, 'n', -3, 15, 11, -3, 8, 9, -3, 2]</code>

◀

Присвоювання частині послідовності порожньої послідовності призводить до вилучення відповідного фрагменту з послідовності, наприклад за виконання

```
s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
s[2:6] = []
print(s)
```

буде виведено

```
[0, 1, 6, 7, 8, 9]
```

Якщо замість **s[2:6] = []** записати **del s[2:6]**, то кінцевий результат не зміниться. За інструкцією вигляду **del s[i]** буде вилучено елемент з індексом **i**.

За виконання **s[len(s):len(s)] = t** елементи з **t** будуть додані в кінець послідовності **s**. Те саме можна записати як **s.extend(t)** або **s += t**.

Додавання елемента **x** в кінець послідовності можна записати як **s.append(x)** або як **s[len(s):len(s)] = [x]**.

Вставку елемента **x** в **i**-ту позицію можна задати як **s.insert(i, x)** або як **s[i:i] = [x]**.

Інструкція **del s[:]** вилучає всі елементи з послідовності **s**. Ту саму дію задає застосування методу **clear**: за виконання виклику **s.clear()** послідовність **s** також стане порожньою.

Метод **pop** дозволяє отримати значення **i**-го елемента та вилучити його з послідовності. Наприклад, з виклику **s.pop(0)** буде повернуто значення елемента з індексом **0** та відповідний елемент буде вилучений. З виклику **s.pop()** буде повернуто значення останнього елемента (**i** відповідний елемент також буде вилучено). За некоректного індексу буде згенеровано виняток типу **IndexError**.

Метод **remove** вилучає з послідовності перше входження заданого елемента, наприклад, за виконання

```
s = [0, 1, 2, 1, 3]
s.remove(1)
print(s)
```

буде виведено

```
[0, 2, 1, 3]
```

У той же час спроба вилучити відсутній у послідовності елемент, наприклад виклик **s.remove(4)**, призведе до генерації винятку типу **ValueError**.

Крім зазначених операцій до змінюваних послідовностей застосовний оператор *****, який задає дописування зазначеної кількості копій у кінець існуючої послідовності.

15.9.2 Деякі особливості вставки та вилучення елементів списків мови Python

Припустимо, що на довгій лавці сидить дев'ять осіб.



Прийшов десятий (червоний) та хоче сісти, наприклад, на позицію з індексом 1. Тоді той, хто знаходиться в цій позиції, та решта за ним мають переміститися на одне місце праворуч.



У наведеному прикладі вставка елемента в послідовність призвела до переміщення лівової частини послідовності. Приблизно те саме відбувається під час вставки елемента в список. Хоч переміщуються не об'єкти, а тільки посилання на них, але і ця операція займає час. Найшвидше виконується операція додавання елемента в кінець списку, бо не потребує переміщень⁵⁴.

Щодо вилучення елементів послідовності, то відбувається зворотній процес: неформально кажучи, елементи після вилученого переміщуються ліворуч, щоб не було прогалів. І, так само, найшвидше виконується операція вилучення елемента з кінця списку, бо не потребує переміщення інших його елементів.

Реалізація списків у поточних версіях Python не передбачає швидкої вставки та вилучення елементів з середини списку. По своїй суті списки Python ближчі до масивів інших мов програмування, але не є їх точним аналогом. Класичний список (не плутати з типом **list**!!!), зазвичай, реалізується на інших засадах і не передбачає операцій прямого доступу, проте допускає швидкі вставку та вилучення елементів у довільну позицію списку.

15.9.3 Зміна порядку елементів послідовності

Найбільш типовими операціями, за виконання яких змінюється порядок елементів послідовності є обернення та сортування послідовності.

Для типу **list**, який є змінюваним, ці операції можуть бути виконані без використання додаткових структур даних, тобто «на місці» (англ. *in place*): результуюча послідовність буде записана на місце вхідної.

Метод **sort** сортує список, а метод **reverse** змінює порядок елементів послідовності на протилежний. Наприклад, за виконання

```
s = [2, 5, 0]
s.sort()
print(s)
```

буде виведено

⁵⁴ Переміщеннями, що виникають у наслідок збільшення/зменшення ділянки пам'яті для збереження послідовності, зараз нехтуємо.

```
[0, 2, 5]
```

Можливе сортування й в оберненому порядку: це регулюється ключовим аргументом **reverse**.
За виконання

```
s = [2, 5, 0]
s.sort(reverse=True)
print(s)
```

буде виведено

```
[5, 2, 0]
```

Продовжуючи далі, за виконання

```
s.reverse()
print(s)
```

буде виведено

```
[0, 2, 5]
```

Незмінювані типи послідовностей методи **sort** та **reverse** не підтримують. Тим не менш серед вбудованих функцій наявна функція **sorted**, яка за послідовністю повертає новий відсортований список. Наприклад, для кортежу **a = (0, 2, 1, 3)** результатом виклику **sorted(a)** буде список **[0, 1, 2, 3]**, а результатом виклику **sorted(a, reverse=True)** список **[3, 2, 1, 0]**.

Вбудований засіб **reversed** за послідовністю будує об'єкт-ітератор, який виконує обхід послідовності в оберненому порядку (*reverse iterator*).

```
a = [0, 2, 1, 3]
ra = reversed(a)
a[1] = 9
print(a) # буде виведено [0, 9, 1, 3]
for i in ra: # буде виведено 3190
    print(i, end="")
```

15.10 Особливості ітерування послідовностями

На поточний момент вже зустрічалися дві конструкції, що дозволяють обходити послідовності. Так, для послідовності **s** можна використати обхід всіх її елементів:

```
for el in s:
    # do smth
    pass
```

або обхід всіх доступних індексів.

```
for i in range(len(s)):
    # do smth with s[i]
    pass
```

Крім зазначених конструкцій ще можна використати бібліотечний засіб **enumerate**, який обходить послідовність, повертаючи пари: індекс елемента та сам цей елемент. Наприклад, за виконання коду

```
s = [1, 3, 7, 8, 9]
for i, el in enumerate(s):
    # do smth with index i and element el
    print(f's[{i}] = {el}')
```

буде виведено

```
s[0] = 1
s[1] = 3
s[2] = 7
s[3] = 8
s[4] = 9
```

Безпечні та небезпечні дії під час обходу послідовності. Нехай задано послідовність *i* в процесі обходу ми хочемо виконати над нею деякі дії.

```
a = [1, 3, 7, 8, 9]
for el in a:
    # do smth
    pass
```

Питання в тім, які операції безпечно виконувати над послідовністю, а які ні. Які вдасться виконати, а які ні. Однозначно, що всі операції, що не змінюють послідовність (наприклад, виводять її елементи, обчислюють якусь статистику за елементами тощо) є безпечними, і видимих перешкод для них нема.

Якщо ми хочемо пройти послідовністю та змінити її елементи, то варто розуміти, що, якщо в циклі присвоїти змінній **el** якесь нове значення, то це ніяк не вплине на значення відповідного елемента послідовності: на кожній ітерації змінна **el** зв'язана з черговим об'єктом, що входить до послідовності, якщо зв'язати її з іншим об'єктом, то це ніяк не вплине на послідовність. Отже, задати перетворення елементів послідовності, ітеруючи циклом саме по елементам послідовності, не вдасться. У цьому випадку слід обходити елементи по індексах, наприклад, так.

```
a = [1, 3, 7, 8, 9]
for i in range(len(a)):
    # do smth with a[i]
    a[i] += 1
```

У наведеному прикладі усі елементи списку будуть збільшені на 1.

Якщо в процесі обходу послідовності ми хочемо додавати або вилучати її елементи, то використання вище наведеної конструкції може призводити до неприємних наслідків (у кращому випадку будуть обійдені не всі елементи, у гіршому випадку — це вже залежить від реалізації — можливе суттєве порушення цілісності пам'яті програми). За необхідності додавати/вилучати елементи варто або розглянути використання циклу **while**, або ітерувати копією послідовності, або перетворювати копію послідовності.

15.11 Послідовності як аргументи функцій

Послідовності, зокрема списки, є контейнерами. Це означає, що функція, аргументом якої є список, здатна змінити значення свого аргументу виклику.

Нижче наведено дві функції. Перша очікує в якості аргументу ціле значення і збільшує отриманий аргумент на 1. Друга функція очікує список і збільшує кожне значення списку на 1.

```
def f1(x):
    x += 1

def f2(x):
    for i in range(len(x)):
        x[i] += 1

a = 1
f1(a)
alist = [1, 5]
f2(alist)
```

За виконання виклику **f1(a)** значення змінної **a** залишається без змін: у тілі функції змінна **x** зв'язується з іншим об'єктом цілого типу. За виконання виклику **f2(alist)** кожен елемент списку **alist** збільшується на 1. Звідси бачимо, що функції можуть змінювати значення своїх аргументів-контейнерів, зокрема у випадку списків. З одного боку, це дає зручний спосіб синтаксичного оформлення модифікацій списків у вигляді функцій. З іншого боку, за неувважності можна змінити значення списку там, де це зовсім не треба.

Також варто згадати, що значення аргументів за умовчанням обчислюються один раз під час виконання інструкції означення функції, а далі використовуються збережені значення. Розглянемо такий **дуже невдалий** фрагмент коду.

Функція **f** приймає аргумент **s**, для якого значенням за умовчанням вказано порожній список. У тілі функції до вхідного списку **s** додається елемент **1**.

```
def f(s=[]):  
    s.append(1)  
    return s
```

Під час виконання виклику функції в інструкції

```
s1 = f()
```

до списку **s** у функції додається елемент **1**. Але, оскільки аргумент виклику не був заданим, то цим списком був список, що є аргументом за умовчанням. Після виконання виклику він отримав значення **[1]**.

Виконаємо ще один виклик.

```
s2 = f()
```

Тепер список, що є аргументом за умовчанням вже має значення **[1, 1]**, бо до нього було додано ще один елемент. Більш того, змінні **s1** та **s2** зв'язані з одним тим самим списком, що є значенням за умовчанням. У цьому легко переконатись, наприклад, виконавши таку інструкцію.

```
print(s1, s1 is s2)
```

За її виконання буде виведено.

```
[1, 1] True
```

Доволі неочікуваний та в більшості випадків небажаний ефект. Щоб його уникнути, в якості значення за умовчанням слід вказати не порожній список, а **None**, та трохи модифікувати код.

```
def f(s=None):  
    if s is None:  
        s=[]  
    s.append(1)  
    return s
```

Тепер кожного разу, коли аргумент не буде задано явно, створюватиметься та повертатиметься новий список, – і така поведінка є більш очікуваною. Так, за виконання

```
s1 = f()  
s2 = f()  
print(s1, s1 is s2)
```

буде виведено таке.

```
[1] False
```

15.12 Оброблення послідовнісних контейнерів: підсумки

Виходячи з використовуваної у мові Python моделі даних, багатовимірні контейнери мають дещо неприємні особливості, пов'язані з неглибоким копіюванням за умовчанням, які було продемонстровано раніше. Неприємні особливості проявляються і за використання послідовнісних контейнерів в якості аргументів функцій, коли функція здатна змінити значення свого аргументу.

Зараз зазначимо лише таке:

- якщо в якості аргументу виклику функція отримує значення змінної, скажімо, **x**, що є кортежем, елементами якого є числа або рядки, або рядком, то незалежно від того, що робить функція зі своїм аргументом, значення змінної **x** по завершенню функції залишається без змін;
- якщо в попередньому випадку значенням змінної **x** є список, то в результаті виконання виклику значення змінної **x** може змінитися;

– якщо значенням змінної `x`, про яку йшлося раніше, є послідовнісний контейнер, окремі елементи якого у свою чергу також є контейнерами, то в результаті виконання виклику значення зображуваних змінною `x` даних може змінитися навіть у випадку, коли `x` є кортежем;

– ітеруючи списком не слід виконувати над ним операції, що змінюють його довжину.

Щодо моделювання математичних векторів та матриць, що виникають у багатьох реальних розрахункових задачах. Так, дійсний вектор можна промодельовати списком дійсних, а матрицю списком, елементи якого у свою чергу є списками (при цьому треба не забути забезпечити цілісність даних; наприклад, щоб усі рядки матриці склалися з однакової кількості елементів). Ці приклади можна побачити в багатьох книжках та підручниках з мови Python. При цьому такі реалізації векторів та матриць важко віднести до ефективних. Не все, що можна зробити, треба робити!

Насправді для мови Python існують спеціалізовані бібліотеки, що містять більш ефективні та доречні засоби для обробки числових векторів та матриць. Найбільш поширена з них — **numpy**.

Незважаючи на це, для задач іншого кола, використання «багатовимірних» списків може бути цілком доречно.

15.13 Оброблення рядків

Об'єкти типів **str**, **bytes**, **bytearray** є послідовнісними контейнерами. Отже, до них застосовні усі операції, що й до послідовнісних контейнерів (з урахуванням змінюваності).

Розглянемо більш специфічні операції, притаманні рядкам.

Зміна та перевірка регістру літер. За рядком можна побудувати рядок, що складається з тих самих літер, але в нижньому регістрі (метод **lower**) або у верхньому регістрі (метод **upper**). Також можна перевіряти, чи правда, що всі літери алфавіту, що входять у рядок є малими (метод **islower**) чи великими (метод **isupper**). Останні дві перевірки ігнорують символи, що не є літерами алфавіту.

'Big brown fox.'.upper()
'BIG BROWN FOX.'
'Big brown fox.'.isupper()
False
'LINUX'.isupper()
True
b'Big brown fox.'.upper()
b'BIG BROWN FOX.'
bytearray(b'Big brown fox.').upper()
bytearray(b'BIG BROWN FOX.')
'Big brown fox.'.lower()
'big brown fox.'

З кириличними літерами це також працює.

'Великий рудий лис №2.'.upper()
'ВЕЛИКИЙ РУДИЙ ЛИС №2.'
'Великий рудий лис №2.'.lower()
'великий рудий лис №2.'
'рудий лис 1234 №2.'.islower()
True

Також існують методи, що перетворюють на велику літеру не всі літери рядка, а тільки першу чи першу літеру кожного слова рядка.

Форматування тексту. Рядок можна центрувати, притискати ліворуч та праворуч. За невказаного символу-заповнювача використовується пробіл.

'Big brown fox.'.center(30)
'Big brown fox.'
'Big brown fox.'.ljust(30,'_')
'Big brown fox._____'
'Big brown fox.'.rjust(30,'-')
'-----Big brown fox.'

Перевірка властивостей літер рядка. Метод **isalpha** перевіряє, чи складається рядок тільки з літер. Білі символи не ігнорує. Символ підкреслення літерою не вважає.

'Рудий лис'.isalpha()
False
'Рудийлис'.isalpha()
True
'Big brown'.isalpha()
False
'Big_brown'.isalpha()
False

Перевіримо, чи складається рядок тільки з символів US-ASCII (з кодами від 0 до 127).

'Big brown fox.'.isascii()
True
'Big brown fox #2.'.isascii()
True
'Великий рудий лис №2.'.isascii()
False

Також наявні методи, що дозволяють перевіряти, чи складається рядок з цифр, з цифр або літер, чи є ідентифікатором.

В останніх версіях Python перевірка належності категоріям символів відбувається відповідно класифікації, заданої Unicode.

Метод **isdecimal** перевіряє, чи складається рядок з десятикових цифр. Десяткові цифри алфавіту хінді, наприклад, ६, також допускаються.

'1234'.isdecimal()
True
'1234६'.isdecimal()
True
int('1234६')
12346

Метод **isdigit** перевіряє, чи складається рядок з цифр, але в типографському розумінні (наприклад, він допустить індекси).

'1234'.isdigit()
True
'1234¾'.isdigit()
False
'1234①'.isdigit()
True

Але при цьому рядок **'1234①'** зображенням цілого числа не є.

Метод **isalnum** перевіряє, чи складається рядок з літер та чисел. Підкреслення він також не допускає.

'1234¾'.isalnum()
True
'лис_2'.isalnum()
False

Також у мові Python є нетипові для бібліотек роботи з рядками методи перевірки чи є рядок ідентифікатором та чи є він зарезервованим словом. Для останньої перевірки треба використовувати засоби модуля **keyword**.

'qwer_ty'.isidentifier()
True
import keyword keyword.iskeyword('qwerty')
False
import keyword keyword.iskeyword('def')
True

Білі символи. З точки зору ANSI C білими (*whitespace*), або пропусками, вважаються символи, що складають рядок **whites**. Тобто не тільки пробіл, а ще табуляція, кінець рядка (перехід на новий рядок), вертикальна табуляція, перехід на нову сторінку, перехід на початок рядка.

whites = ' \t\n\v\f\r'
whites
' \t\n\x0b\x0c\r'

Деякі з білих символів можуть бути задані в рядковому літералі тільки через шістнадцяткові коди, саме тому інтерпретатор показує нам зображення рядка саме так.

Перевіримо, чи складається рядок тільки з білих символів. Для цього використаємо метод **isspace**.

whites.isspace()
True

Крім білих символів таблиці US-ASCII метод **isspace** вважає білими усі білі символи Unicode, наприклад, нерозривний пробіл (*no-break space*, використовується в типографії). Задати його можна через код, який складається з чотирьох шістнадцяткових символів.

'\u00A0' #no-break space
'\xa0'
'\u00A0'.isspace()
True

Вилучення білих символів з кінців рядка.

Вилучимо білі символи з кінця рядка (*r* – з правого боку).

' \t sample \n\v\f\r\t '.rstrip()
' \t sample '

Вилучимо білі символи з початку рядка (*l* – з лівого боку).

' \t sample \n\v\f\r\t '.lstrip()
'sample \n\x0b\x0c\r\t '

А тепер вилучимо білі символи з обох боків рядка.

```
' \t sam ple \n\v\f\r\t '.strip()
'sam ple'
```

Продемонстровані методи дозволяють вилучати не тільки білі символи, а символи із заданого переліку, що розташовуються по кінцях рядка.

```
'Big brown fox.'.rstrip('.x')
'Big brown fo'
```

Починаючи з версії 3.9 наявні методи **removesuffix** та **removeprefix**, які вилучають не перелік символів з кінця або початка рядка, а закінчення чи початок рядка, якщо він співпадає з вказаним.

У всіх випадках результатом є новий рядок.

Пошук та заміна. Рядки підтримують різноманітні пошукові дії.

Перевіримо, чи закінчується рядок на вказане закінчення.

```
'Big brown fox.'.endswith('fox')
False
```

Повторимо перевірку, задавши діапазон початку (**start**) та кінця (**end**) перевірки. Для нумерації позицій використовується звичайна для послідовностей нумерація, де дозволяється від'ємна нумерація. Позиція **end** у діапазон не входить. Якщо її не задати, то буде розглядатися діапазон по кінець рядка включно. За незаданих **start** та **end** пошук відбувається у всьому рядку. Якщо більш точно, то перевірка **s.endswith(substr, start, end)** перевіряє, чи закінчується **s[start:end]** на **substr** (але копіювання частини рядка не відбувається).

```
'Big brown fox.'.endswith('fox',0,-1) #start, end
True
```

Допускається виконувати перевірку кількох закінчень одночасно.

```
'Big brown fox.'.endswith(('fox','fox.'))
True
```

Аналогічну перевірку можна виконувати на початку рядка. Метод **startswith** перевіряє чи є рядок-аргумент початком рядка, для якого викликається метод.

```
'Big brown fox.'.startswith('Big')
True
```

Для цього методу також можна задавати діапазон початку та кінця перевірки

```
'Big brown fox.'.startswith('ig', 1)
True
```

Метод **find** повертає позицію першого входження підрядка в рядок. Пошук відбувається зліва направо. За вказаного діапазону розглядається тільки він, інакше – увесь рядок.

```
'Big brown fox.'.find('fox') #start, end
10
```

Якщо такого підрядка нема, то повертається **-1**.

```
'Big brown fox.'.find('Fox')
-1
```

Але для перевірки входження (коли позиція входження не суттєва) краще використовувати оператор **in**.

```
'Fox' in 'Big brown fox.'
False
```

Метод **index** відрізняється від **find** тільки тим, що за відсутності підрядка генерує виняток.

```
'Big brown fox.'.index('fox')
10
```

```
'Big brown fox.'.index('Fox')
-----
---
ValueError                                Traceback (most recent call
last)
<ipython-input-21-1da121d8353e> in <module>
----> 1 'Big brown fox.'.index('Fox')

ValueError: substring not found
```

У випадку **rfind** (та **rindex**) рядок переглядається від кінця до початку.

```
'Big brown fox fox.'.rfind('fox')    #rindex
14
```

Метод **replace** виконує заміну одного підрядка на інший. Якщо не зазначено інше, то виконуються усі заміни.

```
'1111a1233'.replace('12','2')
'1111a233'
'12121212'.replace('12','3 ')
'3 3 3 3 '
'12121212'.replace('','3')
'31323132313231323'
```

У випадку, коли заміни декілька, пошук наступного входження для заміни відбувається в тій частині вхідного рядка, яка раніше не розглядалась. За цим правилом, якщо після виконання заміни утворюється підрядок, що співпадає з замінюваним, то він не розглядається, бо пошук підрядка, який треба замінити, завжди рухається вперед. Так, у наступному прикладі буде виконано тільки одну заміну.

```
'11111233'.replace('123','23')
'1111233'
```

Можна вказати максимальну кількість заміни, які слід виконати.

```
'12121212'.replace('12','3 ',2)
'3 3 1212'
```

Результатом є новий рядок.

Розбиття та з'єднання рядків. Рядок можна розбити за білими символами. Результатом буде список частин, що утворюються в результаті розбиття. Білі символи до частин не входять.

```
'Big brown fox.\n And once more.'.split()
['Big', 'brown', 'fox.', 'And', 'once', 'more.']
```

Можна вказати максимальну кількість частин, які слід виділити.

```
'Big brown fox.\n And once more.'.split(maxsplit=2)
['Big', 'brown', 'fox.\n And once more.']
```

А також можна розбивати не зліва направо, але й справа наліво.

```
'Big brown fox.\n And once more.'.rsplit(maxsplit=1)
['Big brown fox.\n And once', 'more.']
```

Можна вказати власну послідовність, яку слід використовувати як роздільник.

```
'Big brown fox.\n And once more.'.split(sep='. ')
['Big brown fox.\n And once more.']
```

Варто звернути увагу, що роздільником є сама послідовність-аргумент, а не довільний символ з неї.

За заданого власного роздільника кілька роздільників поспіль як один роздільник не рахуються.

```
'Big brown fox.\n And once more.'.split(sep='.')  
['Big brown fox', '', '\n And once more', '']
```

Якщо аргументи явно не задано, то порожні рядки з результату вилучаються.

```
'Big brown fox.\n And once more.'.split()  
['Big', 'brown', 'fox.', 'And', 'once', 'more.']  
'Big brown fox.\n And once more.'.split(sep=' ')  
['Big', 'brown', '', '', '', 'fox.\n', 'And', 'once', 'more.']
```

У першому випадку усі рядки результату непорожні. У другому випадку обробка виконується дещо інакше, хоча вказаний роздільник є білим символом.

```
''.split()  
[]
```

Можна розбити за послідовністю-роздільником на дві частини. Результатом буде кортеж з трьох рядків: початок рядка, рядок-роздільник, решта рядка, якщо роздільник знайдено, інакше сам рядок та ще два порожніх.

```
'Big brown fox.\n And once more.'.partition(' f')  
('Big brown', ' f', 'ox.\n And once more.')  
'Big brown fox.\n And once more.'.partition('middlepart')  
('Big brown fox.\n And once more.', '', '')
```

Те саме, але тільки пошук роздільника відбувається з кінця рядка.

```
'Big brown fox.\n And once more.'.rpartition(' ')  
('Big brown fox.\n And once', ' ', 'more.')
```

Можна розбити рядок на послідовність логічних рядків. В якості роздільників сприймаються `i \r`, `i \r\n`, `i \n`. Ознаки кінця рядка за умовчанням вилучаються.

```
'Big \n brown fox.\n\r\n And once \r more.'.splitlines()  
['Big ', ' brown fox.', '', ' And once ', ' more.']
```

Але їх можна й залишити.

```
'Big brown fox.\r\n And once \r more.'.splitlines(keepends=True)  
['Big brown fox.\r\n', ' And once \r', ' more.']
```

Можлива і обернена до розбиття операція: за послідовністю рядків можна утворити рядок, що складається з елементів послідовності, записаних через роздільник.

```
'; '.join((str(el) for el in (1,2,6)))  
'1; 2; 6'
```

Використання методу **join** набагато краще з точки зору часу виконання та маніпуляцій пам'яттю програми, бо новий рядок будується "внутрішніми" засобами без численних перебудов.

```
s1 = ['123', 'qw', '2', '45']
s = ''
sep = '; '
for el in s1:
    s += sep + el    #копіювання
```

Рядки є незмінюваними, тому в наведеному циклі на кожній ітерації відбувається копіювання старого рядка в новий з дописуванням символів. Це не тільки працює довше за **join**, але ще вимагає додаткової пам'яті для збереження проміжних результатів.

Рядки типів **bytes**, **bytearray** підтримують схожі операції. Слід пам'ятати, що рядки типу **str** з рядками зазначених типів не сумісні. Рядок типу **bytes** є незмінюваною послідовністю байтів, об'єкт типу **bytearray** зображує змінювану послідовність байтів, а рядок типу **str** є незмінюваною послідовністю символів Unicode.

Рядок типу **bytes** можна задати так.

```
b = b'qwerty'
b
b'qwerty'
```

Вимагається, щоб у запису рядкової константи використовувалися тільки символи ASCII (вони мають коди від 0 до 127). Символи з кодами від 128 до 255 теж є коректними значеннями байтів, але вже мають записуватися через керуючі послідовності. Символи з кодами 256 та більше до складу **bytes** та **bytearray** входити не можуть (бо не є одnobайтовими).

Так утворити зображення рядка байтів у типі **str** можна.

```
str(b)
"b'qwerty'"
```

Але буде отримано не те, що бажалось. Для перетворення байтів на символи слід використовувати метод **decode**.

```
s = b.decode()
s
'qwerty'
```

Для отримання за рядком типу **str** рядка типу **bytes** слід використовувати метод **encode**, який формує відповідну рядку послідовність байтів.

```
s.encode()
b'qwerty'
```

```
'уж'.encode()
b'\xd1\x83\xd0\xb6'
```

Байти зі значеннями 128 та більше під час виведення рядка типу **bytes** відображаються через шістнадцяткові коди.

Для побудови об'єкта **bytearray** слід явно використовувати конструктор.

```
ba = bytearray(b)
ba
bytearray(b'qwerty')
```

В операціях над рядками використовувати одночасно послідовності символів та послідовності байтів не можна. Але типи **bytes** і **bytearray** у більшості випадків сумісні між собою.

```
ba + b'1'
bytearray(b'qwerty1')
```

15.14 Системи із скінченною кількістю станів як засіб моделювання поведінки

Системи із скінченною кількістю станів мають скінченну кількість внутрішніх станів і під дією зовнішніх сигналів можуть змінювати свій стан та видавати певні сигнали на вихід (зокрема, виконувати певні дії). До таких систем можна віднести ліфт, світлофор, банкомат і навіть у деякій мірі різні гаджети.

Банкомат може перебувати в різних станах: очікування, очікування пін-коду, очікування запиту на дію, виконання дії тощо. Виконуваним ним дії залежать від поточного стану. Його функціонування залежить від кількох останніх запитів, а не від усієї історії його роботи. І необхідна для функціонування попередня історія запам'ятовується в стані, яких він має скінченну кількість. Тобто пристрій має *скінченну пам'ять*.

Визначальною характеристикою систем із скінченною кількістю станів є те, що в силу скінченності множини станів вони можуть пам'ятати тільки скінченну історію свого функціонування.

Системи з скінченною кількістю станів, зокрема скінченні автомати, мають певні традиційні застосування:

- 1) пошук у тексті за зразком, де зразок може бути заданий регулярним виразом (англ. *regular expression*);
- 2) обробка текстів у текстових редакторах (наприклад, контекстна заміна);
- 3) лексичний аналіз (використовується не тільки в компіляторах/інтерпретаторах, але й навіть при перетворенні рядків на числові значення);
- 4) моделювання поведінки пристроїв, персонажів в іграх, екземплярів класів, програмних систем тощо, а також моделювання динамічних властивостей програм (керування подіями).

Існують й інші застосування систем із скінченною множиною станів.

Найпростіший шлях побудови розв'язків вказаних вище задач складається в побудові діаграм, що ілюструють структуру зразка, лексем вхідної мови, правил заміни/обробки тощо, а потім перетворення цієї діаграми в алгоритм.

Розглянемо деякі задачі, де виникають системи зі скінченною кількістю станів.

15.14.1 «Розумний» світлофор, або Моделювання фізичного пристрою

Світлофор світить червоним та чекає натискань кнопок, що розташовані по різні боки пішохідного переходу. Якщо натиснуті кнопки з обох боків або тричі з одного, то на 10 с він вмикає зелений та знов починає чекати нові натискання кнопок.

Побудуємо діаграму (див. рис. 46), що висвітлює зміни станів світлофору в залежності від того, які кнопки натиснуті. Натискання кнопки будемо розуміти як сигнал, який отримує система. Зокрема, натисканню кнопки з одного боку зіставимо сигнал l (від *left*), а з іншого — r (від *right*). При розгляді не фізичної моделі світлофора, а його математичної моделі у вигляді системи із скінченною кількістю станів, отримувані сигнали зазвичай називають *вхідними символами*.

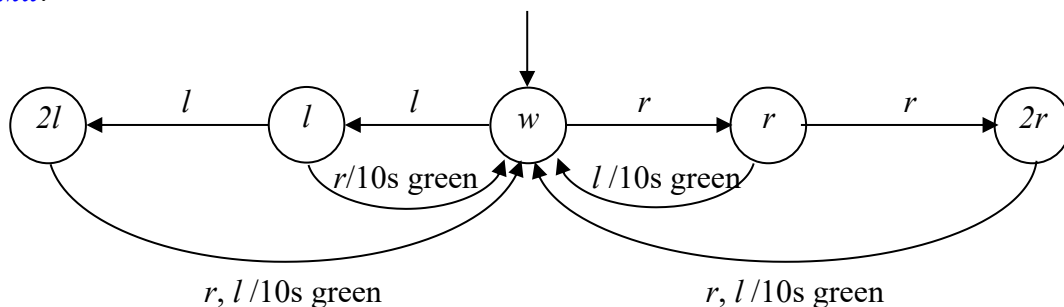


Рис. 46 — Діаграма переходів

Цю діаграму прийнято називати *діаграмою переходів*. Під переходом розуміють перехід системи під дією вхідного символу з одного стану в другий (можливо, той самий).

Діаграма переходів відображає те, як система змінює свій внутрішній стан під дією вхідних символів (сигналів) та, за необхідності, які дії при цьому виконує. Дуга, позначена вхідним символом та, можливо, діями, відображає можливу зміну стану системи під дією вхідного символу. Так, знаходячись у стані *w* під дією вхідного символу *l* система переходить у стан *l* та нічого більше не робить. Далі, зі стану *l* під дією вхідного символу *r* система переходить у стан *w* і при цьому на 10 с перемикає червоне світло на зелене. Стан, в якому система починає свою роботу, позначають стрілкою, що входить в цей стан.

Зауваження. Дуга з міткою "*r, l/10s green*" насправді є графічним скороченням двох дуг з тим самим початком та кінцем але з мітками "*r/10s green*" і "*l/10s green*". Якщо під дією різних символів система переходить з одного стану в інший, то графічно це можна позначити однією дугою з міткою, що перелічує всі символи, за якими можна здійснити цей перехід.

Отже, система починає свою роботу в певному стані (його називають *початковим*), під дією вхідних символів (сигналів, що відповідають натисканню кнопок) вона змінює свій стан та, можливо, виконує певні дії.

Наведена схема дозволяє промодельовувати роботу світлофора.

Суттєво: система пам'ятає лише скінченну інформацію про історію своєї роботи, має скінченну кількість станів, змінює свій стан під дією вхідних символів.

15.14.2 Розроблення алгоритмів за допомогою систем із скінченною кількістю станів

Системи із скінченною кількістю станів дозволяють зручно будувати алгоритми оброблення послідовностей. Розглянемо приклади.

Приклад. Знайти кількість слів у рядку.

Будемо вважати, що словами є довільні непорожні послідовності символів, що відокремлюються роздільниками та не містять роздільників у середині. Для спрощення будемо вважати, що роздільниками є тільки білі символи та коми. Так, рядок '123 1 2, в' містить 4 слова.

Якщо б роздільниками були тільки всі білі символи, або роздільник був єдиним, то для цієї задачі можна було б використати методи **split** або **partition** класу **str**; звідси, якщо рядок не настільки довгий, бо за використання цих методів вміст рядка копіюється в отримувані частини розбиття. Для наведеної множини символів-роздільників використання зазначених методів класу призведе до зайвих не тільки обчислень, але й копіювань інформації. Тому розробимо власний алгоритм, який розв'язує задачу за один прохід по рядку та не копіюючи його окремі слова.

Будемо послідовно переглядати послідовність символів рядка, починаючи з початку слова. Нехай змінна **num** позначає кількість слів у вже обробленій частині слова. Кожне слово рядка будемо рахувати при перегляді його першого символу. На початку роботи встановимо це значення в **0**. Для поточної позиції у слові можна розрізняти два стани: «за межами слова» та «всередині слова». Для позиції перед початком обробки вхідного рядка будемо вважати, що вона знаходиться за межами слова. Якщо поточний символ переводить з-за меж слова всередину слова, то він є першим символом слова й необхідно на 1 збільшити значення змінної **num**.

Альтернативою діаграмі переходів є таблиця переходів, що для кожного стану та для кожного можливого вхідного символу визначає стан, в який має перейти автомат з цього стану за даним вхідним символом. Початковий стан відмічається символом $\rightarrow$.

Іноді при розробці алгоритму буває зручно в таблицю переходів вносити й виконувані під час зміни станів дії.

У нашому прикладі отримаємо таку таблицю 27 зміни станів (та виконуваних дій) залежно від типу поточного символу.

Табл. 27 — Таблиця, що задає роботу шуканого алгоритму

Стан\ Вхідний символ	роздільник	не роздільник
→ за межами слова	за межами слова	всередині слова / збільшити на 1 лічильник слів
всередині слова	за межами слова	всередині слова

Для реалізації алгоритму стан «за межами слова» позначимо як 0, а стан «всередині слова» позначимо як 1.

Для того, щоб у тілі шуканої функції не прив'язуватися до конкретних символів, які можуть бути роздільниками, напишемо функцію **is_delimiter**, що повертає ознаку того, що символ є роздільником. У поточній версії роздільниками вважаємо тільки білі символи та коми, але за потреби за виділення цієї перевірки в окрему функцію множину символів, які мають сприйматися як роздільники, буде легко змінити. Для цього буде достатньо лише змінити код функції **is_delimiter**.

```
def is_delimiter(s):
    """s should be one-symbol string"""
    return s.isspace() or s == ','

def word_count(s):
    # states: 0 - out of a word
    #         1 - inside a word
    state = 0
    num = 0
    for c in s:
        if is_delimiter(c):
            state = 0
        else:
            if state == 0:
                num += 1
                state = 1
    return num
```

Зауваження. Якщо символи треба розділяти не на два класи (роздільники та не роздільники), а на більшу кількість класів, то можна написати функцію, яка за символом буде визначати до якого класу розбиття відноситься символ.

Зауваження. Бувають задачі, в яких фізичний кінець рядка під час побудови таблиці переходів варто розглядати як особливий символ. Відповідні цьому символу дії найчастіше записуються після основного циклу, що обробляє вхідну послідовність.

Приклад: в умовах попереднього прикладу знайти кількість слів довжини 3, що містяться в рядку.

Табл. 28 — Таблиця, що задає роботу шуканого алгоритму

Стан\ Вхідний символ	роздільник	не роздільник
→ за межами слова	за межами слова	всередині слова / лічильник довжини поточного слова встановити в 1
всередині слова	за межами слова / якщо завершилося слово довжини 3, збільшити лічильник таких слів на 1	всередині слова / збільшити на 1 лічильник довжини поточного слова

Запрограмуємо шукану функцію згідно таблиці 28.

```
def word3_count(s):
    # states: 0 - out of a word
    #         1 - inside a word
    state = 0
    num = 0
    cur = 0
    for c in s:
        if is_delimiter(c):
            if state == 1:
                if cur == 3:
                    num += 1
                state = 0
            else:
                if state == 0:
                    cur = 1
                    state = 1
                else:
                    cur += 1
        if state == 1 and cur == 3:
            num += 1
    return num
```

Після циклу записано обробку фізичного кінця рядка, який класифікується як роздільник.

Зауваження. При обробці поточних даних можуть штучно вводити символ, що позначає фізичний кінець послідовності, та обробляти його в загальному циклі.

Приклад. Знайти кількість знакозмін числової послідовності.

Уточнимо, що маємо на увазі під знакозмінною. Якщо послідовність не містить нульових значень, то під знакозмінною будемо розуміти скільки разів сусідні елементи послідовності мають різний знак. Так, послідовність 1, -3, -4, 1, 1, -5 має 3 знакозміни. Якщо послідовність містить нулі, то при підрахунку знакозмін знехтуємо ними (тобто умовно викреслимо їх, а потім порахуємо кількість знакозмін у попередній спосіб). Так, послідовність 1, 0, -1, 0, 1, 2, 0 має 2 знакозміни.

Будемо послідовно переглядати елементи послідовності, починаючи з початку. Нехай змінна **alt** позначає кількість знакозмін у переглянутій частині послідовності. Якщо знак

поточного елемента відрізняється від знаку останнього ненульового його попередника, то знайдено знакозмін (і треба на одиницю збільшити значення змінної **alt**). Тоді у стані будемо пам'ятати знак останнього ненульового попередника (якщо він був) або той факт, що ще не було переглянуто жодного ненульового значення. Отже отримаємо три стани. Позначимо їх **Pos**, **Neg**, **Zero**. На початку роботи не переглянуто взагалі жодного значення, тому в якості початкового стану візьмемо **Zero**.

Алгоритм можна подати такою таблицею 29 зміни станів (та виконуваних дій) у залежності від знаку поточного елемента послідовності.

Табл. 29 — Таблиця, що задає роботу шуканого алгоритму

Стан\ Поточний елемент	додатний	від'ємний	нульовий
→ Zero	Pos	Neg	Zero
Pos	Pos	Neg / збільшити на 1 лічильник знакозмін	Pos
Neg	Pos / збільшити на 1 лічильник знакозмін	Neg	Neg

Запрограмуємо питому функцію згідно таблиці.

```
def alternation_count(seq):
    ZERO = 0
    POS = 1
    NEG = -1
    state = ZERO
    alt = 0
    for el in seq:
        if el < 0:
            if state == POS:
                alt += 1
            state = NEG
        elif el > 0:
            if state == NEG:
                alt += 1
            state = POS
    return alt
```

Наприкінці зазначимо, що програмувати таблицю можна, як спочатку розглядаючи різні випадки вхідних елементів, а потім вже поточний стан (яку наведених прикладах), так і навпаки: спочатку поточний стан, а потім різні випадки вхідних елементів. Якщо таблиця велика та складна, то краще спочатку розглядати випадки різних станів (можливо, виокремлюючи оброблення вхідних елементів в окремі функції, що відповідають станам).

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: структура даних, колекція, контейнер, послідовнісний контейнер, масив, прямий/довільний доступ, послідовний доступ,

індекс, **list**, **tuple**, **len**, **[]**, конкатенація послідовностей, **+**, повторення послідовності, *****, **count**, зріз, **index**, **max**, **min**, неглибоке копіювання, глибоке копіювання, конструктор, ініціалізація, явний виклик конструктора, **append**, **extend**, **insert**, **copy**, **clear**, **pop**, **remove**, **sort**, **reverse**, **sorted**, **reversed**, **+=**, ***=**.

2. У контексті типів, що зображують рядки, дати стислий опис (синтаксис, семантика, особливості): **isspace**, **lstrip**, **rstrip**, **strip**, **lower**, **upper**, **center**, **rjust**, **ljust**, **endswith**, **find**, **rfind**, **in**, **index**, **replace**, **isalpha**, **isascii**, **split**, **rsplit**, **partition**, **rpartition**, **splitlines**, **decode**, **encode**.

3. Вкажіть вбудовані типи послідовнісних контейнерів мови Python. Які з цих типів є змінюваними, а які незмінювані?

4. Які синтаксичні конструкції мови Python призводять до утворення нового об'єкта типу **list**?

5. Які синтаксичні конструкції мови Python призводять до утворення нового об'єкта типу **tuple**?

6. Які операції застосовні до всіх вбудованих послідовнісних контейнерів мови Python?

7. Як індексуються елементи послідовнісного контейнера? Який діапазон чисел для цього застосовний?

8. Що буде за спроби звернутися до елемента послідовнісного контейнера за недопустимим індексом?

9. Як можна перевірити чи входить елемент у числову послідовність, зображену послідовнісним контейнером?

10. Як обчислюються зрізи послідовності?

11. Як оцінюється істиннісне значення послідовності?

12. Яке копіювання відбувається за обчислення зрізу?

13. Якими виразами можна задати додавання елемента в кінець послідовності?

14. Якими виразами можна задати додавання декількох елементів у кінець послідовності?

15. Якими виразами можна задати додавання елемента у визначену позицію послідовності?

16. Якими виразами можна задати додавання декількох елементів у визначену позицію послідовності?

17. Якими виразами можна задати вилучення елемента з послідовності?

18. Якими виразами можна задати вилучення певного відрізка послідовності?

19. Яку семантику має присвоювання зрізу послідовності?

20. Як можна запрограмувати обхід послідовності?

21. Чи можна змінювати послідовність (елементи та/або її довжину), за якою відбувається ітерування? На що слід звертати увагу?

Задачі та вправи

1. Написати код, який виконує такі дії.

Утворити об'єкт списку, що зберігає послідовність 1, 5, 87, 32.

Замінити елемент списку з індексом 1 на число 13.

Додати в кінець списку число 25.

Замінити елемент списку з індексом 1 на послідовність 4, 5.

Вирізати з списку елементи з індексами 3, 4.

Вставити в позицію з індексом 2 число 8.

Побудувати за списком кортеж.

Вивести кортеж.

Додати в кінець списку елементи кортежу.

Вилучити останній елемент списку.

Вивести список.

Вивести кортеж у зворотному порядку.

2. Задано послідовність **s**. Записати вираз, значенням якого є нова послідовність того самого типу, що складається:

- 1) з перших 10 елементів послідовності **s**;
- 2) з перших 10 елементів послідовності **s**, записаних в оберненому порядку;
- 3) з елементів з індексами від 10 до 20 включно;
- 4) з останніх 10 елементів послідовності **s**;
- 5) з останніх 10 елементів послідовності **s**, записаних в оберненому порядку;
- 6) з першої половини послідовності **s** (якщо довжина непарна, центральний елемент включати);
- 7) з першої половини послідовності **s** (якщо довжина непарна, центральний елемент не включати);
- 8) з першої третини послідовності **s**;
- 9) з другої половини послідовності **s** (якщо довжина непарна, центральний елемент включати);
- 10) з другої половини послідовності **s** (якщо довжина непарна, центральний елемент не включати);
- 11) з другої третини послідовності **s**;
- 12) з кожного третього елемента послідовності **s**;
- 13) з кожного третього елемента послідовності **s**, записаних в оберненому порядку.

3. Написати функцію, що виводить елементи послідовності через кому, у кінці має ставитися крапка. Для порожньої послідовності має виводитися повідомлення, що послідовність порожня.

4. Написати функцію, що утворює новий список та заповнює його цілими числами з клавіатури.

а) Завершення введення відбувається тоді, коли чергове значення не може бути перетворено на тип цілих або користувач натисне Ctrl-Z (або Ctrl-D у залежності від ОС).

б) Завершення введення відбувається тоді, коли користувач натисне Ctrl-Z (або Ctrl-D у залежності від ОС). Якщо чергове значення не може бути перетворено на тип цілих, введення вважається неуспішним та повертається **None**.

в) Попередня задача, але за неуспішного введення генерується виняток типу **RuntimeError**.

г) Та сама задача, але за Ctrl-Z (або Ctrl-D у залежності від ОС) введення завершується, а введення нечислових даних ігноруються.

5. Написати функцію, що заповнює вже існуючий список цілими числами з клавіатури.

а) Якщо чергове значення не може бути перетворено на тип цілих, введення вважається неуспішним. Функція повертає ознаку успішності введення. За неуспішного введення вхідний список не змінюється.

б) Попередня задача, але за неуспішного введення вхідний список залишається порожнім.

6. Написати функцію, що утворює новий список та записує в нього **n** цілих чисел, що їх вводить користувач. Число **n** є параметром функції.

а) За неуспішного введення має генеруватися виняток типу **RuntimeError**.

б) Введення нечислових даних ігноруються.

7. Написати функцію, що утворює новий список та заповнює його з клавіатури. Тип елементів списку має бути параметром функції.

8. Написати функцію, що додає в список елемент, якщо його там нема. В усіх випадках повертається індекс елемента.

9. Написати функцію, що утворює новий список та записує в нього **n** різних цілих чисел, що їх вводить користувач. Число **n** є параметром функції. За неуспішного введення має генеруватися виняток типу **RuntimeError**.

10. Написати функцію, що створює список з **n** випадкових чисел.

11. Написати функцію, що виводить усі парні елементи списку цілих.

12. Написати функцію, що в списку цілих замінює всі від'ємні значення на **-1**.

13. Написати функцію, що в списку цілих збільшує всі парні значення на **1**.

14. Написати функцію, що з списку цілих вилучає усі нульові елементи.
15. Написати функцію, що вилучає зі списку усі входження заданого елемента.
16. Написати функцію, що циклічно зсуває список на одну позицію вправо.
17. Написати функцію, що циклічно зсуває список на одну позицію вліво.
18. Написати функцію, що за списком та перестановкою виконує перестановку елементів списку. Додаткові списки не використовувати.
19. Написати функцію, що повертає найменший додатний елемент списку. Якщо додатних елементів нема, має повертатись **None**.
20. Написати функцію, що повертає індекс останнього найбільшого від'ємного елемента списку. Якщо від'ємних елементів нема, має повертатись **-1**.
21. Написати функцію, що повертає кількість елементів списку, що менші своїх сусідів.
22. Написати функцію, що для списку цілих знаходить довжину найдовшого відрізка, в якому всі сусідні числа різні. Якщо таких відрізків кілька, то повернути останній.
23. Написати функцію, що для списку цілих знаходить найдовший відрізок, в якому всі сусідні числа різні. Якщо таких відрізків кілька, то повернути перший.
24. Написати функцію, що за послідовністю цілих та значенням знаходить середнє арифметичне індексів позицій, на яких він зустрічається. За відсутності повернути **-1**.
25. Написати функцію, що за послідовністю цілих знаходить середнє арифметичне її додатніх елементів.
26. Написати функцію, що за послідовністю цілих знаходить середнє арифметичне індексів її додатніх елементів. За відсутності повернути **-1**.
27. Написати функцію, яка за послідовністю чисел $x_0, x_1, \dots, x_{n-1}$ обчислює таку величину:

$$x_{n-1}(x_{n-1} + x_{n-2})(x_{n-1} + x_{n-2} + x_{n-3}) \dots (x_{n-1} + \dots + x_0).$$
28. Написати функцію, яка утворює новий список і заповнює його послідовними біномними коефіцієнтами C_n^k ($k = 0, 1, \dots, n$).
29. Написати функцію, що за заданим цілим n утворює список з елементами

$$C_{2n}^0, C_{2n+1}^1, \dots, C_{2n+k}^k, \dots, C_{3n}^n$$
30. Написати функцію, що перевіряє чи є послідовність:
 - а) неспадною;
 - б) зростаючою;
 - в) монотонною.
31. Розглядаються неспадні послідовності цілих чисел, що зберігаються в списках. Написати функцію, що утворює нову неспадну послідовність, яка є злиттям двох своїх аргументів. Якщо елемент в першу послідовність входить k_1 раз, а в другу — k_2 разів, то в результат він має входити $k_1 + k_2$ разів. Наприклад, за послідовностей 1, 3, 3, 5 та 2, 3, 5, 8 результатом має бути 1, 2, 3, 3, 3, 5, 5, 8.
32. Розглядаються неспадні послідовності цілих чисел, що зберігаються в списках. Написати функцію, що утворює нову неспадну послідовність, яка є об'єднанням двох своїх аргументів. Якщо елемент в першу послідовність входить k_1 раз, а в другу — k_2 разів, то в результат він має входити $\max\{k_1, k_2\}$ разів. Наприклад, за послідовностей 1, 3, 3, 5 та 2, 3, 5, 8 результатом має бути 1, 2, 3, 3, 5, 8.
33. Розглядаються неспадні послідовності цілих чисел, що зберігаються в списках. Написати функцію, що утворює нову неспадну послідовність, яка є перетином двох своїх аргументів. Якщо елемент в першу послідовність входить k_1 раз, а в другу — k_2 разів, то в результат він має входити $\min\{k_1, k_2\}$ разів. Наприклад, за послідовностей 1, 3, 3, 5 та 2, 3, 3, 3, 5, 8 результатом має бути 3, 3, 5.
34. Розглядаються неспадні послідовності цілих чисел, що зберігаються в списках. Написати функцію, що перевіряє чи складаються дві послідовності з однакових елементів. Кількість входжень елемента в послідовність до уваги не брати. Наприклад, послідовності 1, 1, 3 та 1, 1, 1,

3, 3 складаються з однакових елементів. *Обмеження*: використовувати додаткові структури даних для збереження послідовностей або їх частин заборонено.

35. Риби народжуються навесні й живуть не довше 9,5 років. Навесні на кожну рибу припадає в середньому V новонароджених мальків. Кількість риб, незалежно від їх віку, за рік (від весни до весни) зменшується в D разів. Навесні першого року у водоймище випустили M новонароджених мальків. Написати програму обчислення, скільки риб й якого віку буде у водоймищі через Y років.

36. Написати функцію, яка за заданим роком і номером дня в році (від 1 до 365 або до 366, якщо рік високосний) обчислює дату (число, місяць). Наприклад, за 2012 61 і за 2013 60 обчислюється 1 3 — перше березня.

37. Написати функцію, яка за заданою датою (число, місяць, рік, наприклад, 6, 5, 2012) обчислює номер дня в році (від 1 до 365 або 366, якщо рік високосний).

38. Список містить цілі числа. Вилучити зі списку усі повторення (кожне значення має залишитися в одному екземплярі). Використання додаткових структур даних заборонено. Порядок елементів можна змінювати.

39. Попередня задача, але для кожного елемента в списку має залишитись його перше входження і при цьому порядок елементів змінювати не можна.

40. Рядок містить десяткові записи цілих чисел, відокремлені комами, також можуть зустрічатись білі символи. Побудувати відповідний список цілих.

41. За рядком перевірити, чи є він паліндромом.

42. Написати функцію, що вилучає з списку рядків усі рядки з довжиною не більше 5.

43. Написати функцію, що нормалізує список рядків. Під нормалізацією мається на увазі таке: порожні рядки та рядки, що складаються тільки з білих символів, вилучаються; решта рядків стискається (вилучаються білі символи на початку та кінці, усередині довільні послідовності білих символів замінюються на один пробіл).

44. Рядок містить слова, відокремлені білими символами. Знайти кількість слів у рядку.

45. Попередня задача, але рядок дуже довгий та створювати копію навіть одного його слова не можна.

46. Попередня задача, але роздільниками слів є білі символи та знаки пунктуації (.,;:!?).

47. Рядок містить слова, відокремлені білими символами та знаками пунктуації (.,;:!?).

Знайти:

- а) довжину найдовшого слова рядка;
- б) останнє найдовше слово рядка;
- в) кількість слів, довжина яких більша 3;
- г) кількість слів, довжина яких дорівнює 13;
- д) кількість слів, що містять кожну з літер **s, e, l, f**;
- е) кількість слів, що містять **self** як підслово;
- є) останнє слово довжини 13.

Розв'язання оформити у вигляді функції.

Обмеження: копія частини рядка може створюватись тільки якщо вона буде повертатися як результат функції.

48. Написати функцію, що стискає білі символи рядка: білі символи в кінці та на початку рядка вилучаються, у середині рядка кожна послідовність білих символів замінюється на один пробіл.

49. Послідовність задано рекурентним співвідношенням

$$a_{n+1} = a_0 a_n + a_1 a_{n-1} + a_2 a_{n-2} + \dots + a_n a_0 - n, \quad a_0 = 2$$

Написати функцію, що знаходить n -й елемент послідовності.

50. Запрограмувати обчислення функції f , заданої співвідношеннями

$$f(n) = 1, \quad n \leq 3,$$

$$f(n) = f(n-2) + f(n-3), \quad \neg(n : 2), \quad n \geq 4,$$

$$f(n) = 2f(n/2), \quad n : 2, \quad n \geq 4.$$

Використовувати рекурсію заборонено.

51. Розглянемо функцію, що виконує перетворення списку `s`.

```
def f(s, el):  
    while s and s[0] != el:  
        del s[0]
```

а) Яку задачу розв'язує ця функція?

б) Чи можна цю задачу розв'язати більш ефективно? Якщо можна, то як?

52. Розглянемо функцію, що виконує перетворення списку `s`.

```
def f(s, el):  
    i = 0  
    while i < len(s):  
        if s[i] == el:  
            del s[i]  
        else:  
            i += 1
```

а) Яку задачу розв'язує ця функція?

б) Чи можна цю задачу розв'язати більш ефективно? Якщо можна, то як?

53. Розглянемо фрагмент коду, в якому змінна `s` позначає список рядків.

```
res = ''  
sep = '; '  
for el in s:  
    res += sep + el
```

а) Яку задачу розв'язує цей код?

б) Чи можна цю задачу розв'язати більш ефективно? Якщо можна, то як?

16 Рядок виклику та обробка аргументів виклику

16.1 Рядок виклику

Запустити програму на виконання можна не тільки засобами системи програмування або файлового менеджера. Будь-яка операційна система дозволяє запустити програму в так званому **командному рядку**, або **рядку виклику**.

У рядку виклику вказується шлях до файлу з програмою та, можливо, ще декілька слів (послідовностей непорожніх символів) – **аргументів виклику**, що мають оброблятися програмою.

Такі ситуації доволі розповсюджені, наприклад: виклик програми-архіватора, виклик компілятора, виклик інтерпретатора найчастіше здійснюється з аргументами виклику. Зокрема, у виклику

```
python my_prog.py
```

слово **python** задає ім'я виконуваного файлу (система здатна самостійно додати розширення, щоб запустити python.exe), а **my_prog.py** є аргументом виклику, що задає файл з програмою, яку слід виконати.

Операційна система зчитує цей рядок, визначає кількість n слів у ньому й запам'ятовує ці слова у вигляді рядків. Роздільниками слів вважаються білі символи, тому якщо ім'я виконуваного файлу або слова, які записуються в рядку виклику, містять, наприклад, пробіли, то їх слід брати в подвійні лапки, наприклад, **"my program.exe"**. Також існують обмеження на використання в аргументах виклику деяких символів (" , < , > тощо), що мають спеціальне призначення, але зараз не це предмет розгляду.

За рядка виклику

```
program.exe arg1 arg2 arg3
```

операційна система виділить та запам'ятає чотири аргументи виклику:

ім'я виконуваного файлу (ОС бувають різні, але найчастіше буде запам'ятовано повне ім'я, тобто з шляхом включно; наприклад щось таке **C:\MyExe\program.exe**) та ті аргументи, що були записані в командному рядку, тобто **arg1**, **arg2**, **arg3**.

Отримана послідовність рядків передається програмі. Далі програма може її обробляти. Але для цього її слід відповідно написати.

16.2 Обробка аргументів рядка виклику у випадку програми, написаної мовою Python

Коли викликається програма, написана мовою Python, то насправді викликається інтерпретатор Python, якому на вхід дається наша програма, і система виконує з рядком виклику все, що має.

Дуже бажано, щоб аргументи виклику потрапляли до нашої програми, а не залишалися тільки для інтерпретатора (хоча деякі з них, записані перед іменем ру-файлу, завжди призначаються саме інтерпретатору).

Розглянемо, що від рядка виклику

```
python testargv.py 1 2 3
```

отримає наша програма, записана у файлі **testargv.py**. Інтерпретатор залишить їй ім'я ру-файлу (може бути записано повний шлях), та решту аргументів виклику, записаних після нього. З цих рядків він сформує список, який буде доступним через змінну **sys.argv** (змінна **argv** з модуля **sys**). Щоб обробити в програмі аргументи виклику, необхідно імпортувати модуль **sys**.

Наведемо код програми **testargv.py**, яка вміє обробляти власні аргументи виклику.

```
import sys
print('The number of arguments:', len(sys.argv))
for i,arg in enumerate(sys.argv):
    print('argv['+i,']= ', arg, sep='')

print('The type of sys.argv is',type(sys.argv))
print('The type of sys.argv[0] is',type(sys.argv[0]))
```

testargv.py

За наведеного вище рядка виклику буде виведено

```
The number of arguments: 4
argv[0]= D:/testargv.py
argv[1]= 1
argv[2]= 2
argv[3]= 3

The type of sys.argv is <class 'list'>
The type of sys.argv[0] is <class 'str'>
```

16.3 Стандартні потоки

Під час виконання програма взаємодіє з оточуючим середовищем, у тому числі через стандартні потоки вводу-виводу. Зазвичай, стандартний вхід відбувається з клавіатури, а стандартний вихід виводиться на монітор.

Насправді, стандартних потоків не два, а три і вони мають номери.

0 — стандартний вхідний потік (**stdin**), за умовчанням дані в нього надходять з клавіатури; саме з стандартного вхідного потоку відбувається введення за виклику **input()**;

1 — стандартний вихідний потік (**stdout**), за умовчанням його вміст відображається на моніторі; за виклику **print("something")** виведення відбувається саме в нього;

2 — стандартний потік помилок (**stderr**), за умовчанням його вміст також відображається на моніторі; у цей потік, наприклад, потрапляють діагностичні повідомлення інтерпретатора щодо неперехоплених винятків.

Обробник командного рядка дає можливість перенаправляти ці стандартні потоки.

Так, за виклику

```
my.exe <input.txt
```

в якості стандартного вхідного потоку буде використано файл **input.txt**.

За виклику

```
my.exe >output.txt
```

стандартний вихідний потік буде перенаправлено у файл **output.txt**.

За виклику

```
my.exe 2>output.txt
```

стандартний потік помилок буде перенаправлено у файл **output.txt**. Ліворуч від знаку > можна вказувати номер стандартного потоку. Якщо номер не вказано, то підставляється 1.

За виклику

```
my.exe 2>err.txt 1>out.txt <in.txt
```

стандартний вхідний потік буде асоційовано з файлом **in.txt**, стандартний вихідний потік – з файлом **out.txt**, стандартний потік помилок – з файлом **err.txt**.

За виклику

```
my.exe 1>out.txt 2>&1
```

стандартний вихідний потік та стандартний потік помилок будуть асоційовані з одним тим самим файлом **out.txt**. Запис **2>&1** буквально означає таке: спрямувати стандартний потік помилок туди, куди спрямовано стандартний вихідний потік.

У випадку використання перенаправлення > результуючий файл, якщо його не було, то створюється; якщо існував, то стає порожнім. Якщо необхідно дописувати результати в кінець файлу, то слід використовувати перенаправлення >>, за якого очищення існуючого файлу відбуватися не буде. За послідовних викликів

```
my1.exe >out.txt
```

```
my2.exe >>out.txt
```

у файлі **out.txt** буде накопичуватися стандартний вихід від роботи програми **my1.exe** та **my2.exe**.

При цьому частина рядка виклику, що відповідає перенаправленням, до аргументів виклику не додається. Тобто для виклику

```
c:\my.exe arg1 arg2 2>err.txt 1>out.txt <in.txt
```

кількість аргументів виклику дорівнюватиме 3 і цими аргументами будуть рядки **c:\my.exe**, **arg1** та **arg2**.

За виклику з командного рядка

```
python testargv.py 1 2 3 >out.txt
```

список **sys.argv** буде так само містити 4 рядки, частина **>out.txt**, що відповідає за перенаправлення потоків, до аргументів командного рядка не потрапить.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: рядок виклику, командний рядок, **stdin**, **stdout**, **stderr**, <, >, >>, &, **sys.argv**.

2. Які елементи можуть бути записані в рядку виклику?
3. Як операційна система обробляє рядок виклику?
4. Для чого при запису рядка виклику можуть використовуватись подвійні лапки?
5. В який спосіб під час запуску можна перенаправити вхід чи вихід програми?
6. Чи має програма доступ до аргументів рядка виклику, яким була запущена на виконання?
7. Змінною якого типу зображується рядок виклику в мові Python? Який тип у мові Python мають його окремі аргументи?
8. Що є першим аргументом рядка виклику, який отримує програма?

Задачі та вправи

1. Запустити довільну власну програму, що виконує обчислення якого-небудь виразу, написану на лабораторних заняттях, спрямувавши її вихід у файл 1.txt.

2. Ще раз запустити програму (див. попередню задачу) на виконання, спрямувавши її вихід у файл 1.txt у режимі дописування.

3. Створити текстовий файл, що містить вхідні дані для програми (див. попередні дві задачі), та запустити її на виконання, підставивши створений вхідний файл замість консольного входу.

4. Написати програму, яка відповідно до вмісту командного рядка виконує певні дії. Мають підтримуватися команди:

pi — вивести число π з 4-ма знаками після крапки;

e — вивести число e з 6-ма знаками після крапки;

а також команда **help**, за якою виводиться довідка по допустимих для програми аргументах командного рядка.

За незрозумілого вмісту командного рядка виводиться відповідне повідомлення та довідка по аргументах командного рядка.

Зверніть увагу: у процесі роботи програми користувач з консолі жодних команд вводити не повинен.

Протестувати написану програму, запустивши її з різними рядками виклику.

17 Вбудовані засоби роботи з послідовностями

17.1 Ітератори

17.1.1 Поняття ітератора

Під **шаблоном проєктування** можна розуміти типове розв'язання типової проблеми. Адже, замість того, щоб кожного разу розв'язувати задачу з нуля, більш ефективно використати апробований раніше підхід. Сучасне програмування уявити без шаблонів просто неможливо.

Концепція ітератора спочатку виникла як шаблон проєктування. Згодом ітератори стали невід'ємною частиною бібліотек, що реалізують контейнери даних (колекції).

Для контейнера типовою є задача перегляду всіх його елементів. Наприклад, роздрукувати всі дані, що містяться в контейнері, знайти суму всіх елементів тощо. У випадку вбудованих послідовнісних контейнерів задача здається надуманою, адже для них можливий прямий доступ до елементів за їх номером. Насправді, не всі, навіть послідовнісні контейнери (існують не тільки вбудовані, але й інші, причому побудовані на дещо інших «фізичних» принципах ☺), надають операцію прямого доступу за індексом. Тоді задача обходу суттєво ускладнюється. А якщо треба організовувати ще кілька обходів одночасно, наприклад, як для такої задачі:

для кожного елемента послідовності знайти кількість менших за нього елементів, що розташовані після нього; у кінці обробки надрукувати найбільший такий елемент, для якого цей показник є найменшим,

то ситуація стає ще більш складною. Відкривати клієнтам контейнера його внутрішню будову є грубим порушенням принципів інкапсуляції та абстракції даних. Крім іншого це змусить користувача контейнера вникати в деталі його реалізації, що може потребувати відповідної алгоритмічної підготовки. Додавати відповідний метод для кожного такого алгоритму теж хибний підхід. Заздалегідь передбачити всі алгоритми неможливо, а записувати в означенні класу алгоритми для специфічних задач не дуже правильно. Тим більш, що обов'язком структури даних є збереження даних, а не реалізація алгоритмів над нею.

Поміркуємо, що саме задає індекс, за яким обходимо, скажімо список типу **list**, і які він має можливості. На перший погляд можна сказати, що він задає позицію і його можна збільшити на 1. Якщо зазирнути глибше, то можна казати, що він задає поточний елемент списку і дає можливість від поточного елемента перейти до наступного за ним. Маючи можливість виконувати ці дві операції, а також позиціонуватись на початок, клієнту контейнера вже не треба піклуватися про внутрішній устрій структури даних, яку він використовує. Ітератори можна вважати засобом задання позиції у послідовності даних контейнера.

Ітератори дозволяють працювати з різними за внутрішньою будовою контейнерами в однаковий спосіб (згадаємо: з точки зору використання цикл **for** однаково обходить і списки, і кортежі, і рядки (і багато інших структур даних), а ґрунтується він саме на ітераторах). У найпростішому випадку від ітератора вимагається давати можливість послідовного обходу контейнера (наприклад, відвідування всіх її елементів). Як правило, ітератор забезпечує доступ на читання поточного елемента (якщо такий є) та вміє переходити до перегляду наступного елемента контейнера (у порядку обходу). Також має бути якась ознака завершення обходу.

17.1.2 Різновиди ітераторів

Ітератори бувають різні (правда, стандартних бібліотек Python це не дуже стосується).

З точки зору можливих операцій над елементами, ітератор може давати не тільки доступ на читання поточного елемента, але й повний доступ (поточний елемент можна не тільки прочитати, але й змінити). Бувають ітератори, що під час обходу виконують перетворення елементів.

З точки зору напрямку обходу, ітератор може бути не тільки однонаправленим (рухається тільки в одному напрямку), але й двонаправленим (рухається і вперед, і назад).

Також для ітераторів можуть вводити спеціальні значення, які позначають значення за кінцем послідовності, яку обходить ітератор (англ. *past-the-end values*). Ці значення ітераторів використовуються алгоритмами обходу як ознака того, що обхід завершено.

Для ітераторів можуть мати сенс операції порівняння (наприклад, чи встановлено ітератори на одну ту саму позицію) та знаходження відстані. Маючи операцію порівняння ітераторів, можна казати про обхід частини контейнера, що задається як **[from, to)**, де **from, to** — значення ітераторів.

```
current = from
while current!=to:
    переставити current на наступний елемент
```

Саме ітератори є однією з причин того, що позиція **to** до діапазону не входить: якщо порівняння на рівність для ітераторів означено завжди, то порівняння **<**, **<=**, **>**, **>=** для ітераторів означають не завжди (для деяких структур даних підтримка таких порівнянь потребує значних зусиль, у той же час перевірки **==** та **!=** реалізуються просто).

17.1.3 Ітератори в мові Python

Повернемося до реалій мови Python, яка концепцію ітераторів підтримує, але в доволі спрощеному варіанті.

Згадаємо цикл **for**, який можна використовувати для обходу різних контейнерів.

```
s = [1, 6, 7]
for el in s:
    інструкції
```

Насправді він реалізується саме за допомогою ітераторів.

Схема виконання **for** за ітерування по непорожній структурі даних виглядає приблизно так.

1) Створюється об'єкт ітератора, встановлений на початок списку **s**
it = iter(s)

Для цього використовується вбудована функція **iter**.

2) Отримання поточного елемента та встановлення ітератора на наступний елемент реалізовано як одну операцію, яку здійснює вбудована функція **next**. Коли всі елементи контейнера вже пройдено, то за спроби отримати черговий елемент генерується виняток типу **StopIteration**. Поява цього винятку є ознакою того, що обхід завершено.

```
while True:
    try:
        el = next(it)
    except StopIteration:
        break
    інструкції
```

Зауваження. Остання конструкція наведена виключно для того, щоб показати, що відбувається насправді. У більшості ситуацій перевагу слід віддавати циклу **for**.

У мові Python від ітераторів вимагається таке⁵⁵:

1) Значенням функції **iter** на ітераторі має бути той самий ітератор, який є аргументом функції.

Зауваження 1. Це потрібно для того, щоб, наприклад, у циклі **for** можна було вказувати не тільки структуру даних, але й заздалегідь встановлений ітератор. Зокрема, для обходу в оберненому порядку може використовуватися **reversed**, що повертає відповідним чином налаштований ітератор, не змінюючи при цьому дані, які обходяться.

Зауваження 2. Виклик функції **iter** не утворює копію ітератора. Для копіювання ітераторів використовуються набагато складніші механізми (і буде правдою сказати, що штучні).

⁵⁵ Насправді вимоги накладаються на службові методи класів ітераторів.

2) Результатом виклику для ітератора функції **next** є черговий елемент послідовності (також переноситься поточна позиція, на яку встановлено ітератор), якщо ітератор обхід не завершив. Якщо обхід послідовності завершено, то генерується виняток типу **StopIteration**.

3) Якщо внаслідок виклику для ітератора функції **next** було згенеровано виняток типу **StopIteration**, то всі подальші виклики **next** для цього ітератора також мають завершуватись винятком типу **StopIteration**.

Якщо значенням функції **iter** для об'єкта є ітератор, то *по такому об'єкту можна ітерувати* (англ. *iterable object*). Зокрема, ітерувати можна по спискам, кортежам, рядкам тощо.

Функція **iter** визначена для об'єктів не всіх типів: якщо для об'єкта її значення не визначене, то за виклику виникає помилка. Наприклад, ітерувати не можна за цілими та дійсними числами.

Приклади

1. Нехай є список

```
s = [1, 6, 7]
```

Встановимо ітератор на його початок.

```
i = iter(s)
```

Отриманий об'єкт є ітератором.

```
type(i)
```

```
list_iterator
```

За ітератором можна запустити цикл **for**: цикл обійде всі значення, що ще не були пройдені ітератором. У прикладі це буде весь список.

```
for el in i:  
    print(el, end=' ')
```

```
1 2 6
```

Спозиціонуємось на початок ще раз

```
i = iter(s)
```

та отримаємо перший елемент послідовності.

```
next(i)
```

```
1
```

Якщо тепер запустити цикл **for**, то він пройде тільки частину списку від позиції, заданої ітератором. Перший елемент був отриманий раніше, тому в циклі він отриманий вже не буде.

```
for el in i: print(el, end=' ')
```

```
6 7
```

Тепер обійдемо список у зворотному порядку. Для цього використаємо **reversed**, що повертає відповідний ітератор.

```
reversed(s)
```

```
<list_reverseiterator at 0x2e5abd52e50>
```

```
for el in reversed(s): print(el, end=' ')
```

```
6 2 1
```

2. Перевіримо, чи дійсно по завершенню обходу функція **next** генерує виняток.

```
s = [1, 6, 7]
```

```
next(i)
```

```
1
```

Зараз отримано перший елемент.

```
next(i)
```

```
2
```

Отримали другий.

```
next(i)
```

```
6
```

Отримали третій.

Більше елементів у послідовності нема. За спроби отримати наступний елемент

```
next(i)
```

```
-----  
---  
StopIteration                                Traceback (most recent call  
last)  
<ipython-input-60-a883b34d6d8a> in <module>  
----> 1 next(i)  
  
StopIteration:
```

отримуємо виняткову ситуацію.



17.1.4 Об'єкти, які можна викликати

Вбудована функція **iter** повертає ітератор, що задає обхід об'єкта. Деталі, пов'язані з класами, залишимо на потім, а зараз познайомимось з іншою цікавою концепцією мов програмування, яка може бути використана для побудови ітераторів.

- Під **об'єктом, який можна викликати** (англ. *callable object*), розуміють об'єкти, які можна викликати як функцію (з чи без аргументів — неважливо).

У Python до такої категорії об'єктів відносяться функції, але не тільки вони⁵⁶.

Крім розглянутого вище випадку, ітератор може бути побудований за виконання виклику **iter(callable_obj, sentinel)**. У цьому разі для об'єкта **callable_obj** має бути коректним вираз **callable_obj()**. Отриманий ітератор за кожним зверненням за черговим елементом здійснює виклик **callable_obj()** і, якщо отримане значення дорівнює **sentinel**, генерує виняток типу **StopIteration**. Тобто значення **sentinel** задає ознаку того, що ітерування слід завершити.

Приклад. Побудуємо ітератор, який обходить псевдовипадкову послідовність цілих від 0 до 4 включно. Псевдовипадковою будуть не тільки елементи послідовності, але й її довжина.

Намагаючись отримати імітацію випадкових чисел, було написано численні алгоритми, що імітують послідовності випадкових чисел. Ззовні здається, що послідовність дійсно випадкова, але насправді працюють цілком конкретні, детерміновані алгоритми, що за станом процесу детерміновано, без жодної випадковості визначають наступний елемент. Початковий стан генератора задається так званим «зерном», від якого відштовхується побудова послідовності.

У Python наявний модуль **random**, що містить засоби для отримання випадкових чисел. Під час його імпортування відразу відбувається ініціалізація стану генератора, тому вручну встановлювати зерно не треба. Зараз в якості зерна використовується системний час, а тому використання модуля **random** у реальних проєктах, пов'язаних з криптографією, є небезпечним.

Напишемо функцію **f**, що повертає псевдовипадкове ціле число від 0 до 5 включно.

```
import random  
# random.seed() виконується під час ініціалізації модуля, тут є зайвим  
def f():  
    return random.randrange(0,6)
```

Функція **randrange(start, stop)** повертає випадкове ціле в діапазоні **[start, stop)**.

⁵⁶ Крім функцій, наприклад, можна викликати об'єкти класів, для яких перевантажено «магічний» метод **__call__**, а також і об'єкти, що зберігають самі класи (наприклад, можливий запис **int(3)**).

Тепер з використанням функції **iter** побудуємо за функцією **f** ітератор, задавши в якості значення, на якому ітератор має зупинитись ціле **5**, та проітеруємо.

```
for el in iter(f,5):  
    print(el, end=' ')
```

```
2 1 2
```

17.1.5 Можливості функції **next**

Для ітератора **it** з виклику **next(it)** повертається поточне значення, на яке вказує ітератор, при цьому ітератор переставляється на наступний елемент послідовності. Якщо поточного значення нема (обхід завершено), то генерується виняток **StopIteration**.

Виклик **next(it, value)** відрізняється від попереднього тільки тим, що якщо чергового значення нема, то з виклику повертається значення **value**.

Приклад використання функції **next** з двома аргументами.

Задамо список та влаштуємо за ним ітерацій на одну більше, ніж в ньому елементів. В якості другого аргументу для функції **next** вкажемо число **13**.

```
s = [1, 2]  
it = iter(s)  
for _ in range(len(s) + 1):  
    print(next(it, 13), end=' ')
```

```
1 2 13
```

Згідно запрограмованому спочатку було здійснено обхід існуючих елементів, за спроби взяти третій елемент замість згенерованого винятку отримали число **13** — другий аргумент функції **next**.

Цей приклад виключно штучний, тільки для демонстрації можливостей функції **next**. На практиці так обходити структури даних не слід.

17.1.6 Копіювання ітераторів

Копіювання ітераторів у мові Python **не передбачено**. Проте засоби, що дозволяють це зробити для ітераторів контейнерів, існують.

Приклад. Перш за все переконаємось, що функція **iter** копію ітератора не утворює (принаймні для класів стандартної бібліотеки). Для цього встановимо ітератор **i1** на початок списку **s**, а ітератор **i2** отримаємо за допомогою **iter** та ітератора **i1**.

```
s = [1, 2, 3, 6]  
i1 = iter(s)  
i2 = iter(i1)  
print(i1 is i2)
```

```
True
```

Як можна бачити з прикладу, новий об'єкт утворений не був.

Тобто далі ітератори **i1** та **i2** будуть рухатись послідовністю одночасно.

Якщо далі отримаємо поточні елементи кожного з ітераторів, то отримаємо два послідовні елементи списку.

```
a = next(i1)  
b = next(i2)  
print(a, b)
```

```
1 2
```

Якщо виникає гостра потреба скопіювати ітератор, наприклад, в якому алгоритмі необхідно запам'ятати поточну позицію, то можна виконати копіювання ітератора на рівні копіювання двійкового вмісту об'єкта. Для цього можна використати модуль **pickle**.

Функція **dumps** повертає значення типу **bytes**, в яке записує послідовність байтів, що відповідає об'єкту. Обернена функція — **loads**, за двійковим значенням будує об'єкт з цим значенням.

Приклад копіювання ітераторів

```
import pickle

s = [1, 2, 3, 6]
i1 = iter(s)
next(i1)

1
```

Зараз ітератор **i1** встановлено десь в середині послідовності. Створимо інший ітератор, що буде вказувати на ту саму позицію послідовності.

```
i2 = pickle.loads(pickle.dumps(i1))
print(i1 is i2)

False
```

Тепер ітератори **i1** та **i2** дійсно різні та рухатимуться послідовністю незалежно.

Спочатку вони встановлені на один той самий елемент послідовності.

```
a = next(i1)
b = next(i2)
print(a,b)

2 2
```

Змістимо **i1** на кілька позицій.

```
next(i1)

3
```

```
next(i1)

6
```

А тепер пересунемо **i2**.

```
next(i2)

3
```

Як можна бачити з прикладу, пройдена ітератором **i1** частина послідовності не вплинула на обхід ітератора **i2**.



Слід розуміти, що можливість скопіювати ітератор з використанням засобів модуля **pickle** існує далеко не для всіх ітераторів і залежить від деталей реалізації ітератора, які не визначаються мовою. Тому, виконуючи таке копіювання ітератора, як в прикладі, ми покладаємось на незадокументовані можливості мови Python, що робити **не варто**.

17.2 Наповнювання, або абстракція, списків

Нехай нам необхідно побудувати список, що складається з квадратів перших 10 натуральних чисел. Можна використати такий фрагмент коду.

```
s = []
for i in range(10):
    s.append(i * i)
```

За його виконання значенням змінної **s** буде список⁵⁷ **[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]**. Проте мова Python має більш зручний механізм, який тут може бути використаний. У результаті

⁵⁷ Так, число **0** вважається натуральним.

виконання наступної інструкції змінна `s` отримає те саме значення, що й у попередньому випадку.

```
s = [i*i for i in range(10)]
```

Цей механізм має назву **наповнювання**, або **абстракції**, **списку**, або **list comprehension**. Він дозволяє використовувати для побудови списку ітерування (**for**) та умови (**if**).

Зазвичай вираз наповнювання виконується швидше за код, що в циклі послідовно додає елементи.

Зауваження щодо перекладу. Переклад терміну «*list comprehension*» не є усталеним та однозначним. Серед тих інтернет-джерел, що не використовують англійську транскрипцію замість перекладу, розповсюдженим є переклад **абстракція списку**, що походить з російськомовного перекладу класичної книги з функціонального програмування.

Абстракція в математиці. Множину можна задати явним переліком її елементів, наприклад, $\{1, 2, 3\}$, або можна задати так $\{n \mid n \in N, 1 \leq n \leq 3\}$ — абстрактно.

У тій книзі термін «абстракція списку» позначав конструкцію функціональної мови програмування Miranda, що в компактний спосіб задавала списки. Якщо не занурюватися в подробиці, то список означався в термінах певних операцій над списками, у тому числі в якості аргументів та результатів цих операцій допускалися нескінченні списки, тобто список задавався саме *абстрактно*. Якщо до цього ще додати той факт, що в суто функціональних мовах програмування програма складається з означень, а її виконання є обчисленням функції (причому без жодних побічних ефектів), то переклад «абстракція списку» у той книзі був доречним і повною мірою відображав зміст відповідної синтаксичної конструкції.

Також у тому перекладі наявний синонім — «включення» списків: новий список утворювався фільтрацією та перетворенням елементів вхідного списку, і цей синонім теж інколи згадується як переклад.

У документації на Python поруч з «*list comprehension*» можна зустріти «display» у тому розумінні, що *list comprehension* є одним із механізмів задання списку⁵⁸ (тобто зображення в тексті програми). Звідси в інтернеті можемо знайти інший варіант перекладу — «подання списків»⁵⁹. Щодо «генераторів списків» (і таке теж можна зустріти), то термін «генератор» у мові Python має своє власне значення, а тому такий переклад явно не дуже вдалий.

Виходячи зі значень слова *comprehension* — охоплення, повнота, зміст, у цьому тексті для **list comprehension** буде вживатися переклад **наповнювання списку**.

Дещо змінимо задачу: побудуємо список квадратів перших 10 натуральних чисел, що не діляться на 3. Не використовуючи синтаксичні можливості Python код мав би вигляд.

```
s = []
for i in range(10):
    if i%3:
        s.append(i*i)
```

Проте за допомогою наповнювання списку те саме значення `[1, 4, 16, 25, 49, 64]` змінній `s` можна надати однією інструкцією.

```
s = [i*i for i in range(10) if i%3]
```

Наповнювання списку є виразом, що задає список. У найпростіших випадках цей вираз може мати вигляд

1) **[*expr* for *variable* in *iterable*]**, де

expr — вираз, що задає значення, яке додається до списку;

iterable — вираз, по значенню якого можна ітерувати;

variable -- змінна, значенням якої є поточний елемент об'єкта, по якому ітеруємо.

⁵⁸ Інший механізм — явний перелік елементів списку у квадратних дужках.

⁵⁹ У цьому варіанті ще неправильно перекладено слово «display», це доволі типова помилка невідповідного використання слова «подання». Згідно тлумачних словників, «подання — офіційна письмова заява з проханням до вищестоящої інстанції».

У цьому прикладі він складається з єдиної секції **for**, що має вигляд **for variable in iterable**.

2) [expr for variable in iterable if condition],

Тут вказано секцію **for** і після неї секцію **if**, яка має вигляд **if condition**. У цьому випадку до списку додаються тільки ті значення виразу **expr**, під час отримання яких виконувалася умова **condition**.

Область дії означення змінної **variable** обмежується виразом наповнювання. Під час обчислення наповнювання завжди створюється нова змінна **variable**, яка на час обчислення перекриває зовнішнє ім'я **variable** (якщо воно було). По завершенню обчислення утворена змінна припиняє своє існування.

Ім'я **variable** можна використовувати в **expr**, а також у всіх секціях після секції **for**, в якій вона позначає поточне значення.

Приклад.

```
i = 2020
s = [i * i for i in range(10)]
print(s)
print(i)
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
2020

Обчислення виразу наповнювання на значення змінної **i** не вплинуло, бо область дії означення змінної, що позначає поточний елемент, обмежена виразом.



У загальному випадку кількість вкладених секцій **for** та **if** може бути більшою.

Приклади

1. За наявності кількох секцій **for**, найлівіша відповідає зовнішньому циклу, решті відповідають послідовно вкладені цикли.

```
s = [(i, j) for i in range(0, 2) for j in range (3, 5)]
s
```

[(0, 3), (0, 4), (1, 3), (1, 4)]

Це відповідає приблизно такому.

```
s = []
for i in range(0,2):
    for j in range (3,5):
        s.append((i,j))
```

2. У другій секції можна задіяти змінну, уведену в першій секції.

```
s = [(i, j) for i in range(0, 3) for j in range (i, 3)]
s
```

[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]

3. Умови можуть йти всуміш з секціями **for**.

```
s = [(i, j) for i in range(0, 2) if i!=1 for j in range(3, 4) if j!=4]
s
```

[(0, 3)]

4. Наступний вираз можливий і побудує те саме значення, що і в попередньому прикладі. При цьому він дуже неоптимальний за часом виконання. Адже в ньому внутрішній цикл буде виконуватись навіть для тих значень **i**, де його можна було б не виконувати.

```
s = [(i, j) for i in range(0,2) for j in range (3,4) if i!=1 and j!=4]
s
```

[(0, 3)]

5. Кілька умов також можуть йти поспіль. Знайдемо числа, що не діляться на жодне з чисел 2 та 3.

```
s = [i for i in range(0, 20) if i%2 if i%3]
s
[1, 5, 7, 11, 13, 17, 19]
```

6. Вирази наповнювання можуть бути вкладеними.

```
s = [[ 1 if i==j else 0 for j in range(4)] for i in range(3)]
for el in s:
    print(el)
[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 0, 1, 0]
```

Зазначимо, що тут зовнішній цикл відбувається по **i**, а внутрішній по **j**, оскільки для кожного значення **i** обчислюється вкладений вираз наповнювання.

◀

Як бачимо з останнього прикладу, утворення вкладених списків також можливе. Крім того, конструкція наповнювання застосовна не тільки до списків, але й до множин та словників, про що буде далі.

- Складних та важко зрозумілих виразів наповнювання слід уникати, віддаючи перевагу більш традиційним конструкціям.

17.3 Генератори

Робота з ітераторами переважно розуміється як отримання послідовності значень за деякою вже існуючою в пам'яті сукупністю даних. На отримання послідовності значень можна подивитися інакше: замість того, щоб читати існуючі дані, будемо генерувати елементи шуканої послідовності. Обидва підходи задають обхід послідовності, але в другому випадку можна казати, що послідовність отримується «на льоту». Зрозуміло, що це економить пам'ять.

Ідея генераторів полягає в тому, щоб видавати значення за запитом. Аналогічно ітераторам, з виклику **iter(g)** для генератора **g** має повертатися значення **g**, а з виклику **next(g)** — черговий елемент послідовності, а коли послідовність вичерпано, генеруватися виняток **StopIteration**.

Формально, генератори мови Python також є і ітераторами. За генераторами мови Python можна ітерувати, тобто їх можна використовувати так само як ітератори, хоча вони мають додаткові можливості (розглядатись не будуть).

Внутрішня будова генераторів набагато складніша. По суті вони є сопрограмами ($\approx$ функціями, що вміють призупиняти власне виконання та видавати проміжні результати). Їх копіювання не є можливим, навіть за допомогою модуля **pickle**.

17.3.1 Вирази, що задають генератори

Раніше розглядали приклади утворення списків за допомогою наповнювання (*comprehension*). Замість того, щоб явно утворювати список, можна побудувати генератор, який буде послідовно утворювати елементи цього списку, не зберігаючи список у пам'яті. Для цього у виразі наповнювання достатньо змінити квадратні дужки на круглі.

Розглянутий раніше вираз

```
[i for i in range(0, 20) if i%2 if i%3]
```

задає утворення списку цілих від 0 до 19 (включно), що не діляться на жодне з чисел 2 та 3. При цьому список явно будується та зберігається в пам'яті.

Вираз

```
(i for i in range(0, 20) if i%2 if i%3)
```

задає утворення генератору, що за запитом (викликом функції **next**), видає черговий елемент тієї самої послідовності, але при цьому всю послідовність у пам'ять не записує.

Приклади

1.

```
g1 = (i for i in range(0, 20) if i%2 if i%3)
g1
<generator object <genexpr> at 0x000002E5ABD71350>
```

```
for el in g1:
    print(el, end=' ')
1 5 7 11 13 17 19
```

2. Щоб «на льоту» отримати послідовність, що утворюється із заданої поелементним застосуванням перетворення, можна використати вираз, що задає генератор.

```
g = (i*3+2 for i in range(5))
g
<generator object <genexpr> at 0x0000022023B7DA50>
for el in g:
    print(el, end=' ')
2 5 8 11 14
```

17.3.2 Інструкція *yield*

Інший спосіб утворити генератор — використати інструкцію **yield**.

Розглянемо функцію, що обчислює послідовність чисел Фібоначчі та використовує інструкцію **yield**.

```
def fibo_ranged(count):
    a, b = 1, 0
    i = 0
    while i < count:
        yield b
        a, b = b, a + b
        i += 1
```

Та виконаємо цикл.

```
for el in fibo_ranged(10):
    print(el, end=' ')
0 1 1 2 3 5 8 13 21 34
```

Якщо уважно подивитись на код, то виведена послідовність елементів є тією послідовністю, яка отримується за виконання інструкції **yield b** під час виконання циклу.

Розберемось, що відбувається насправді.

Нехай маємо функцію **f**, в якій наявна інструкція **yield**.

```
def f(...):
    ...
    yield вираз
    ...
```

Перш за все, ім'я **f** позначає функцію. Але через наявність інструкції **yield** ця функція має особливу поведінку. У документації такі функції називають *функціями-генераторами* (англ. *generator function*).

За виконання виклику **f(...)** виконання тіла функції **f** розпочато **не** буде. Натомість буде утворено об'єкт-генератор, типом якого є вбудований клас **generator**. Цей об'єкт будується за функцією **f** і далі буде керувати її виконанням.

Нехай виконано інструкцію **g = f(...)**, тобто ім'я **g** позначає щойно створений генератор, який ще не розпочав виконання своєї функції-генератора.

За виконання виклику **next(g)** розпочнеться виконання тіла функції. Як тільки потік виконання досягне інструкції **yield вираз**, то після обчислення виразу виконання функції **f** призупиниться, а обчислене значення буде повернуто як результат виклику.

За наступного виконання виклику **next(g)** виконання функції **f** продовжиться з тієї точки, на якій було призупинено. При цьому стан пам'яті функції (її локальні змінні) будуть зберігатися в об'єкті-генераторі, а отже під час призупинення локальні змінні функції **f** будуть зберігати свої значення.

Якщо за виконання виклику **next(g)** виконання функції **f** буде завершено, то об'єкт генератора згенерує виняток типу **StopIteration**.

Наведена семантика генераторів дає можливість по них ітерувати, як це було продемонстровано в початковому прикладі.

Приклад. Подивимось на прикладі, що маємо у випадку функцій-генераторів.

Як зазначалося, ім'я **fibo_ranged** позначає функцію.

```
fibo_ranged
<function __main__.fibo_ranged(count)>
```

За виконання виклику цієї функції утворюється генератор. Обчислення функції у цей момент ще **не** розпочато.

```
fibo = fibo_ranged(1)
fibo
<generator object fibo_ranged at 0x0000022023ABA350>
```

За виконання виклику функції **next** розпочинається виконання функції-генератора і з інструкції **yield** отримується значення.

```
next(fibo)
0
```

На наступному виклику виконання продовжується з точки зупинки. Цикл завершується і виконання тіла функції **fibo_ranged** також завершується. На цьому генератор генерує виняток типу **StopIteration**.

```
next(fibo)
-----
---
StopIteration                                Traceback (most recent call
last)
<ipython-input-6-c1c75ce7f1a4> in <module>
----> 1 next(fibo)

StopIteration:
```

До об'єктів-генераторів застосовні методи **close**, **send** та **throw**. Методи **send** та **throw** дають можливість передавати інформацію у функцію-генератор. Використання цих методів⁶⁰ виходить за межі початкового знайомства з мовою Python, а тому в цьому посібнику розглядатись не буде. Метод **close** детально розглядатися теж не буде, але про його існування корисно знати: він зупиняє процес виконання функції-генератора, не чекаючи, поки ця функція буде завершена у звичайний спосіб. Оскільки генератор зберігає стан пам'яті своєї функції-генератора, то його виклик може забезпечити більш оперативне вивільнення ресурсів (пам'яті тощо).

Приклад нескінченного генератора.

У математиці є таке поняття: актуальна нескінченність. Під цим терміном мають на увазі, що в кожен скінченний момент часу ми можемо отримати тільки скінченну сукупність елементів, але потенційно ми можемо отримати їх скільки завгодно: треба 100 — можемо отримати 100, треба 10^{100} — можемо отримати і стільки.

⁶⁰ У багатопотоковому програмуванні для взаємодії потоків використовують так звані асинхронні генератори, які взаємодіють між собою за допомогою методів **send** та **throw**.

Наведена нижче функція-генератор потенційно здатна згенерувати довільну кількість чисел Фібоначчі.

```
def fibo_inf():
    a, b = 1, 0
    while True:
        yield b
        a, b = b, a + b
```

Згенеруємо, скажімо, перші 15 чисел Фібоначчі.

Створимо об'єкт-генератор та використаємо його в циклі на 15 ітерацій.

```
fibo = fibo_inf()
for _ in range(15):
    print(next(fibo), end=' ')
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Оскільки виконання функції **fibo_inf** передбачає нескінченний цикл, то конструкція **for el in fibo_inf()** є помилковою, бо за відсутності переривання циклу призведе до зациклювання або інших неприємностей.

Оскільки після виконаного ітерування виконання функції-генератора не завершилося, то варто зупинити її виконання за допомогою виклику.

```
fibo.close()
```

За однією функцією-генератором можна одночасно мати кілька утворених генераторів. Значення локальних змінних функції-генератора не будуть розділятися між різними генераторами, натомість кожен генератор буде зберігати значення змінних саме того виконання функції, яким він керує.

Якщо утворити два генератори, то можна побачити, що це різні об'єкти.

```
g1 = fibo_inf()
g2 = fibo_inf()
print(g1, g2, sep='\n')
g1.close()
g2.close()

<generator object fibo_inf at 0x0000022023B6BA50>
<generator object fibo_inf at 0x0000022023ABA190>
```

Якщо зазирати ще глибше, то буде зрозуміло, що й виконуючі структури в них також різні (бо за різними адресами).

Інструкція **yield** може мати вигляд **yield from вираз**. У цьому випадку значення виразу розглядається як вкладений ітератор (англ. *subiterator*) і за чергового виконання **yield from** повертається отримане від ітератора значення. Якщо ітератор вичерпано, то виконуються наступні за **yield from** інструкції.

Приклад з вкладеним ітератором.

```
def f():
    yield from [1, 2, 3]
    print('next portion')
    yield from [4, 5]

for el in f():
    print(el, end=' ')
```

```
1 2 3 next portion
4 5
```

Бачимо, що спочатку елементи генеруються за списком **[1, 2, 3]**, далі продовжується виконання функції (виводиться повідомлення) і після елементи генеруються за списком **[4, 5]**.

Ще один невеликий **приклад**, щоб наочно продемонструвати, що за створення генератора виконання функції не розпочинається.

```
def f():
    print("let's go")
    yield from [1, 2, 3]

g = f()
```

Нічого виведено не буде.

```
el = next(g)
let's go
```

А ось на першому зверненні до `__next__` розпочинається виконання функції-генератора та отримується значення, яке далі виведемо в інтерпретаторі.

```
el
1
```

Будемо чемними та завершимо виконання функції-генератора.

```
g.close()
```

- Якщо виконання функції-генератора було розпочато, але не завершилось у звичайний спосіб або за винятком, то для відповідного генератора слід викликати метод **close**.

17.4 Лямбда-вирази та анонімні функції

Іноді виникає потреба в невеликих функціях, які треба одноразово використати. З цією метою можна створювати **анонімні функції** — функції, що в програмі не мають власного імені.

У використанні між звичайними (означеними за допомогою інструкції означення функції) та анонімними функціями принципової різниці нема.

Результатом обчислення **лямбда-виразу**, тобто виразу вигляду:

lambda перелік параметрів через кому: вираз

є анонімна функція, яку ще називають **лямбда-функцією** (англ. *lambda function*). Лямбда-функція має вигляд.

```
def lambda(перелік параметрів через кому):
    return вираз
```

Анонімні функції зручно використовувати під час сортування.

Приклад. Припустимо, що необхідно відсортувати список пар. Це нескладно зробити, використовуючи метод **sort**. При цьому пари будуть сортуватись лексикографічно, тобто спочатку по першій координаті, потім по другій.

```
li = [(3, 15), (2, 1), (3, 10), (3, 1), (1, 10)]
li.sort()
li
[(1, 10), (2, 1), (3, 1), (3, 10), (3, 15)]
```

Уявимо, що список необхідно сортувати, беручи до уваги тільки другу координату. Метод **sort** дозволяє передати функцію, яка буде застосовуватись до кожного елемента перед порівнянням і далі вже порівнюватимуться перетворені елементи. Тоді, щоб відсортувати тільки по другій координаті пар, слід передати функцію, що для пари повертає її другу координату. Для цього в наступному фрагменті використовується лямбда-функція.

```
li = [(3, 15), (2, 1), (3, 10), (3, 1), (1, 10)]
li.sort(key=lambda x: x[1])
li
[(2, 1), (3, 1), (3, 10), (1, 10), (3, 15)]
```

А тепер відсортуємо так, щоб спочатку пари упорядковувалися за другим елементом, а в межах однакового другого — за першим. Математично це задається таким співвідношенням.

$$(x_1, y_1) \leq (x_2, y_2) \Leftrightarrow (y_1 < y_2) \vee ((y_1 = y_2) \wedge (x_1 \leq x_2))$$

Для цього використаємо функцію, що під час сортування буде переставляти перші два елементи кортежу.

```
li = [(3, 15), (2, 1), (3, 10), (3, 1), (1, 10)]
li.sort(key=lambda x: (x[1], x[0]))
li
[(2, 1), (3, 1), (1, 10), (3, 10), (3, 15)]
```

Анонімні функції зручні не тільки для сортування. Коло їх застосувань значно ширше. Наприклад, значення за умовчанням параметрів-функцій зручно задавати саме як анонімні функції.

Приклад. Напишемо функцію, що має забезпечити введення дійсного числа, що задовольняє умові. Якщо користувач вводить послідовність символів, що не може бути перетворена на дійсне, або число не задовольняє умові, то має генеруватись виняток. Замість будови системи власних класів винятків повідомлення про причину проблеми будемо передавати у винятку класу **Exception** (альтернативний варіант: доповнювати перехоплений виняток власною діагностичною інформацією та передавати далі).

Спочатку забезпечимо введення дійсного числа. Функція **input_float** реалізує введення дійсного числа користувачем, у випадку помилок вона генерує виняток. Параметр **prompt** використовується як запрошення на введення.

```
def input_float(prompt=None):
    try:
        x = input(prompt)
    except Exception:
        raise Exception('There are no input to convert to float.')

    try:
        x = float(x)
    except ValueError:
        raise Exception(f'Input [{x}] could not be converted to float')

    return x
```

Тепер додамо перевірку умови. Функція **input_with_check** буде реалізовувати введення дійсного числа користувачем, але вже з перевіркою умови. Щоб дозволити не вказувати у виклику запрошення **prompt** на введення, і другий формальний параметр повинен мати значення за умовчанням. В якості такого значення доцільно використати функцію, що завжди повертає істинне значення. (І навіть якщо б не синтаксична необхідність, то все одно мало сенс для функції, що перевіряє коректність значення, надати значення за умовчанням). Ця функція може бути записана як анонімна, щоб не захаращувати код допоміжними іменами та щоб зекономити час на розробку.

```
def input_with_check(prompt=None, check=lambda x: True):
    x = input_float(prompt)
    if not check(x):
        raise ValueError('Incorrect value')
    return x
```

Викликати цю функцію можна з анонімною функцією, що задає перевірку, якщо ця перевірка нескладна.

```
try:
    eps = input_with_check('Введіть епсілон >0: ', check=lambda x: x>0)
except Exception as e:
    print(e)
```

```
Введіть епсілон >0: 0
Incorrect value
```

Якщо перевірка задається складним виразом чи підставляється неодноразово, то краще створити окрему іменовану функцію та використовувати її.

Анонімні функції використовують, коли над послідовністю треба виконати нескладне перетворення її елементів (приклади будуть далі). Ще кілька випадків використання анонімних функцій пов'язані з обробниками подій у графічних інтерфейсах. Зрозуміло, що наведений перелік вичерпним не є.

- Якщо функція нескладна, згадується в тексті програми разово та самостійного значення не має, то можна розглянути використання замість неї анонімної функції.
- Анонімними функціями захоплюватись не варто.

17.5 Вбудовані засоби роботи з послідовностями

Багато бібліотечних функцій здатні працювати не тільки з кортежами, списками, множинами тощо, а з довільними об'єктами, по яких можна ітерувати. Крім об'єктів-контейнерів до таких об'єктів відносяться і «чисті» ітератори, і генератори, та не тільки.

На прикладах розглянемо кілька вбудованих функцій, що використовують/породжують об'єкти, за якими можна ітерувати.

Функція **sum**, аргументом якої є об'єкт, по якому можна ітерувати, повертає суму елементів, що їх обходить ітератор.

Функції **min**, **max** для аргументу, по якому можна ітерувати, повертають відповідно найменше/найбільше значення його елементів. У цьому випадку можна задати ключовий аргумент **default** — значення, що буде повертатись за порожньої послідовності. Застосовані для сукупності аргументів, функції **min**, **max** повертають найменше/найбільше значення відповідно.

Для функцій **min**, **max** можна задавати ключовий аргумент **key**, який має той самий зміст, що і випадку сортування: це має бути функція, що застосовується до елементів перед порівнянням.

Приклади.

```
s1 = (1, 3, -7, 2, 6)
print(sum(s1), min(s1), max(s1))
```

```
5 -7 6
```

```
min(s1, key=lambda x: abs(x))
```

```
1
```

```
min((), default=13)
```

```
13
```

```
print(min(1,4,7), max(2,7,4))
```

```
1 7
```

Нехай у нас є відображення $f: A \rightarrow B$, яке перетворює елементи множини A . Доволі природно розширити його на послідовності, що складаються з елементів множини A , застосовуючи до кожного елемента окремо. Відповідне розширення можемо задати так.

Нехай задано послідовність

$$a = (a_1, a_2, \dots, a_n), a_i \in A, i \in \overline{1, n},$$

покладемо за означенням

$$f(a) = (f(a_1), f(a_2), \dots, f(a_n)).$$

Таке перетворення послідовностей є доволі розповсюдженим і в бібліотеках мов програмування існують засоби, що його реалізують. У мові Python таким засобом є **map**.

У більшості джерел **map** вважають функцією, що виконує перетворення послідовності значень. Якщо більш точно, то **map** є класом, а результатом обчислення виразу **map(f, s)** є об'єкт, обхід якого дає послідовність, що отримується в результаті поелементного застосування функції **f** до елементів з **s**.

У виразі **map(f, s1, s2)** функція **f** має бути застосовна до двох аргументів, які одночасно беруться з послідовностей **s1** та **s2**. Більша кількість послідовностей-аргументів також можлива: у такому разі функція **f** повинна приймати стільки аргументів, скільки задано послідовностей.

Приклади.

1. За послідовністю **s1**

```
s1 = (1, 3, 7, 2, 6)
```

побудуємо послідовність квадратів її елементів, використовуючи **map**. Обчислення квадрату задамо лямбда-функцією

```
it = map(lambda x: x*x, s1)
```

З виклику буде повернуто об'єкт, за яким можна ітерувати. Виведемо елементи отриманої послідовності.

```
for el in it:
    print(el, end=' ')
```

```
1 9 49 4 36
```

2. Знайдемо суму квадратів елементів послідовності **s1**.

Це можна зробити через вираз, що задає генератор.

```
sum(x*x for x in s1)
```

```
99
```

Або з використанням **map**.

```
sum(map(lambda x: x*x, s1))
```

```
99
```

У цій ситуації **map** можна було уникнути. Але, якщо перетворення має застосовуватись до кількох значень одночасно, то використання **map** призведе до більш читабельного виразу з меншою кількістю штучних перетворень порівняно зі спробою виразити те саме без **map**.

Об'єкти класу⁶¹ **filter** реалізують вибірку елементів послідовності **s** згідно заданої умови **f**. Послідовність **s** має задаватися об'єктом, за яким можна ітерувати. Умова **f** має задаватися функцією (або чимось іншим, що можна викликати).

Послідовність, отримана за об'єктом **filter(f, s)**, може бути задана виразом **(el for el in s if f(el))**. Якщо значенням **f** є **None**, то результат відповідає виразу **(el for el in s if el)**.

Приклади.

⁶¹ Саме так, це не функція, а клас.

1. Знайдемо елементи послідовності **s1 = (1, 3, 7, 2, 6)**, що націло діляться на 3.

```
g = filter(lambda x: x%3==0, s1)
for el in g:
    print(el, end=' ')
```

3 6

2. Якщо функцію-фільтр не задано, то залишатимуться елементи, які оцінюються як істинні.

```
for el in filter(None, range(5)):
    print(el, end=' ')
```

1 2 3 4

Вбудована функція **any** за аргументом, по якому можна ітерувати, перевіряє, чи є серед елементів, що переглядаються під час обходу, істинні. Математичним аналогом є квантор існування.

Вбудована функція **all** за аргументом, по якому можна ітерувати, перевіряє, чи всі елементи, що переглядаються під час його обходу, є істинними. Математичним аналогом є квантор загальності.

За допомогою виразу, що задає генератор, та функцій **any** та **all** можна швидко запрограмувати перевірку умов для сукупності значень.

Нехай задано послідовність **s**.

```
s = (1, 3, 5, 7, 9)
```

У загальному випадку послідовність може бути задана не тільки кортежем, а довільним об'єктом, за яким можна ітерувати.

Перевіримо, чи правда, що послідовність містить парні числа. Для цього перетворимо послідовність на послідовність ознак непарності її елементів. Необхідності зберігати отриману послідовність у пам'яті нема, тому використаємо вираз, що задає генератор. Оскільки він буде єдиним аргументом виклику, то дужки навколо нього ставити не потрібно.

```
any(el%2 == 0 for el in s)
```

False

Перевіримо, чи правда, що всі елементи непарні.

```
all(el%2 for el in s)
```

True

Перевіримо, чи є всі елементи більшими за 1.

```
all(el > 1 for el in s)
```

False

Клас⁶² **zip** дає потужний апарат для перетворення послідовностей. Деякі речі з використанням об'єктів цього класу виглядають просто та очевидно, але деякі доволі нетривіальні і не сприяють легкому сприйняттю коду.

Нехай

$s_1, \dots, s_k$ позначають послідовності,

n – є найменшою з довжин цих послідовностей,

вираз $s_i(j)$ позначає j -ий елемент послідовності s_i (нумерація елементів починається з 0),

тоді

zip($s_1, \dots, s_k$) задає ітератор, що реалізує обхід послідовності

(($s_1(0), s_2(0), \dots, s_k(0)$), (($s_1(1), s_2(1), \dots, s_k(1)$), ..., (($s_1(j), s_2(j), \dots, s_k(j)$), ...,

⁶² Так, і **zip** також не є функцією.

$(s_1(n-1), s_2(n-1), \dots, s_k(n-1))$,

що має довжину n та складається з k -елементних кортежів. Елементи результуючої послідовності утворюються з відповідних елементів послідовностей $s_1, \dots, s_k$.

Якщо послідовностей дві, то результуюча послідовність складається з пар, а сам процес дуже нагадує застібку-блискавку, звідки й походить назва.

Об'єкт класу **zip** можна побудувати за довільними об'єктами, по яких можна ітерувати. Ітерування по отриманому об'єкту припиняється як тільки буде вичерпано хоча б одну вхідну послідовність (залишки решти залишаються необробленими). Починаючи з Python 3.10 є можливість генерувати виняток, якщо з'єднуються послідовності різних довжин.

Приклади.

Розглянемо послідовності **s1** та **s2**.

```
s1 = (1, 3, 7, 2, 6)
s2 = [5, 7, 8, 0]
```

1. За допомогою **zip** можна одночасно ітерувати кількома послідовностями.

З'єднаємо послідовності **s1** та **s2** за допомогою **zip** та проітеруємо по результату.

```
for el in zip(s1, s2):
    print(el, end=' ')
(1, 5) (3, 7) (7, 8) (2, 0)
```

Бачимо: перша послідовність мала довжину 4, друга – довжину 5. Отже, результуюча послідовність має довжину 4 – найменшу з довжин послідовностей, за якими будувалася.

2. Зробимо ще одне з'єднання, але використаємо одну ту саму послідовність кілька разів.

```
for el in zip(s1, s2, s1):
    print(el, end=' ')
(1, 5, 1) (3, 7, 3) (7, 8, 7) (2, 0, 2)
```

Оскільки **s1** є не ітератором, а контейнером, то під час побудови результуючого ітератора, обходи першої та третьої вхідної послідовності виконуються незалежно (бо для контейнера **s1** з викликів **iter(s1)** повертаються різні ітератори, що обходять **s1**).

3. За допомогою **zip** послідовність кортежів можна розділити на кортеж послідовностей.

Розглянемо послідовність пар.

```
pairs = [(1, 2), (3, 4), (5, 6), (7, 8)]
```

Якщо **s** є послідовністю, то вираз ***s** задає її розпакування: підстановку замість єдиного об'єкту-послідовності послідовності її об'єктів-елементів.

Якщо виконано **s2 = [5, 7, 8, 0]**, то вираз **print(*s2)** еквівалентний такому **print(5, 7, 8, 0)**.

Тут виводиться послідовність **s2** як єдине ціле.

```
print(s2)
[5, 7, 8, 0]
```

А тут виводяться окремі елементи **s2**.

```
print(*s2)
5 7 8 0
```

За розпакування послідовності у виразі **zip(*pairs)** отримаємо вираз **zip((1, 2), (3, 4), (5, 6), (7, 8))**, за яким має відбутися з'єднання чотирьох послідовностей. Результатом має бути послідовність **((1, 3, 5, 7), (2, 4, 6, 8))**, що складається з четвірок і має довжину 2.

```
x, y = zip(*pairs)
print(x)
print(y)
(1, 3, 5, 7)
(2, 4, 6, 8)
```

Бачимо, що за послідовністю пар було утворено дві послідовності з першими та другими координатами цих пар.

4. Розглянемо трохи складніший приклад розділення послідовностей на окремі координати.

Станом на зараз вираз `zip(s1, s2)` задає послідовність пар (1, 5), (3, 7), (7, 8), (2, 0). Підставимо це значення у вираз `zip(*zip(s1, s2))`. За розпакування послідовності отримаємо вираз

`zip((1, 5), (3, 7), (7, 8), (2, 0))`

За правилами результатом останнього виразу має бути послідовність, що складається з четвірок, що отримуються одночасним обходом кожної з вхідних послідовностей, тобто послідовність, що складається з двох послідовностей: ((1, 3, 7, 2), (5, 7, 8, 0)).

```
print(s1)
print(s2)
x, y = zip(*zip(s1, s2))
print(x)
print(y)

(1, 3, 7, 2, 6)
[5, 7, 8, 0]
(1, 3, 7, 2)
(5, 7, 8, 0)
```

Можна бачити, що виконане перетворення з точністю до відкинутих через різну довжину елементів, відновило початкові послідовності.

Можливо, наведене перетворення виглядає красиво. Але є певні нюанси:

- під час розпакування `*zip(s1, s2)` фізично будувався кортеж аргументів;
- і `x`, і `y` у цьому прикладі є кортежами, тобто фізично будуються в пам'яті.

З наведеного випливає, що для довгих послідовностей, це не кращий спосіб виділення окремих координат їх елементів.

5. Використання `zip` дозволяє групувати сусідні елементи послідовності.

Побудуємо ітератор `it`, який обходить цілі числа від 0 до 18 включно.

```
it = iter(range(19))
```

Тепер з'ясуємо, що є значенням виразу `*(it,)*3`. Вираз `(it,)` задає одноелементний кортеж, що складається з побудованого ітератора. Вираз `(it,)*3` задає кортеж, що утворюється як конкатенація трьох копій кортежу `(it,)`, тобто його значенням є `(it, it, it)`. Вираз `*(it,)*3` задає розпакування кортежу на окремі елементи, які далі віддаються функції. Отже, виклик `zip(*(it,)*3)` може бути переписаний як `zip(it, it, it)`, що дозволить легше зрозуміти, що буде відбуватися далі.

Під час конструювання `zip`-об'єкта

```
g = zip(*(it,)*3)
```

для кожного аргументу виклику буде викликано функцію `iter`. Але аргументи є ітераторами, отже функція `iter` поверне для кожного з них той самий об'єкт. Якщо помітити той факт, що на місце кожного аргументу підставлявся один той самий об'єкт-ітератор, то для кожної вхідної послідовності `zip`-об'єкт буде використовувати один той самий ітератор.

Під час ітерування за `zip`-об'єктом для кожного з трьох отриманих ітераторів буде братися черговий елемент. Але! Ці ітератори задаються одним тим самим об'єктом, тому на кожній ітерації з однієї тієї ж самої послідовності буде вибиратися чергова трійка елементів. Що можна чітко побачити, виконавши наступний код.

```
for el in g:
    print(el, end=' ')

(0, 1, 2) (3, 4, 5) (6, 7, 8) (9, 10, 11) (12, 13, 14) (15, 16, 17)
```

На останньому кроці елементів на трійку не вистачило, тому числа 18 не бачимо.

◀

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: шаблон проєктування, ітератор, генератор, функція-генератор, вираз, що задає генератор, **iter**, **next**, **StopIteration**, **yield**, **yield from**, **all**, **any**, **sum**, **min**, **max**, **map**, **filter**, **zip**, лямбда-вираз, анонімна функція, лямбда-функція, вираз наповнювання списків.

2. Що означає, що об'єкт підтримує ітерування (тобто, що по ньому можна ітерувати)?

3. Об'єкти яких вбудованих класів підтримують ітерування?

4. Як у мові Python відбувається обхід за допомогою ітераторів?

5. Для чого та як можна використовувати ключовий аргумент **key** методу **sort** (а також аналогічної функції)?

Задачі та вправи

1. Написати вираз, значенням якого є список, що складається:

а) з послідовних цілих чисел від 2 до 19 включно;

б) з квадратів послідовних цілих чисел від 1 до 10 включно;

в) з квадратів послідовних цілих чисел від 10 до 0 включно;

г) з цілих чисел від 0 до 100 включно, які не діляться на жодне з чисел 2, 3, 5.

2. Написати вираз-генератор, що задає послідовність, що складається:

а) з квадратів послідовних цілих чисел від 1 до 10 включно;

б) з квадратів послідовних цілих чисел від 10 до 0 включно;

в) з цілих чисел від 0 до 100 включно, які не діляться на жодне з чисел 2, 3, 5.

3. Написати вираз, що для дуже довгої послідовності цілих чисел задає підпослідовність її елементів, що

а) діляться на 2;

б) не діляться на жодне з чисел 2, 3, 5;

в) більші за задане число;

г) є додатними;

д) є простими;

е) є точними квадратами.

4. Написати вираз, що для дуже довгої послідовності чисел задає підпослідовність її елементів, що:

а) є числами типу **int**;

б) є числами типу **float**.

в) не є числами типу **float**;

г) не є рядками типу **str**.

5. Написати вираз, що для дуже довгої послідовності цілих чисел задає підпослідовність, що складається:

а) з першої половини послідовності (якщо довжина непарна, центральний елемент включати);

б) з першої половини послідовності (якщо довжина непарна, центральний елемент не включати);

в) з першої третини послідовності;

г) з другої половини послідовності (якщо довжина непарна, центральний елемент включати);

д) з другої половини послідовності (якщо довжина непарна, центральний елемент не включати);

е) з другої третини послідовності;

є) з кожного третього елемента послідовності;

ж) з кожного третього елемента послідовності, записаних в оберненому порядку.

Обмеження: утворювана послідовність повністю в пам'яті збережена бути не може.

6. Написати вираз, значенням якого є кількість додатних елементів числової послідовності s .
7. Написати вираз, значенням якого є кількість входжень десяткових цифр у рядок.
8. Написати вираз, значенням якого є послідовність позицій, на яких елемент **el** зустрічається в послідовності s .
9. Написати вираз, значенням якого є найменший додатний елемент послідовності або **None**, якщо таких елементів у послідовності нема.
10. Написати вираз, значенням якого для послідовності цілих чисел є сума усіх її парних елементів. Кожне парне число враховується стільки разів, скільки входить у послідовність.
11. Написати вираз, значенням якого для послідовності є сума усіх її числових елементів. Кожне число враховується стільки разів, скільки входить у послідовність.
12. Написати вираз, значенням якого для послідовності цілих чисел є сума її елементів, що займають позиції
 - а) з парними індексами;
 - б) з непарними індексами.
13. Написати вираз, що перевіряє, чи правда, що всі елементи числової послідовності:
 - а) є додатними;
 - б) є цілими.
14. Написати вираз, що перевіряє, чи правда, що всі елементи послідовності цілих:
 - а) діляться на 7;
 - б) більші за задане число;
 - в) є точними квадратами;
 - г) є простими числами.
15. Написати вираз, що перевіряє, чи правда, що послідовність цілих містить:
 - а) число 7;
 - б) від'ємні числа;
 - в) хоча б одне число, що ділиться на 13.
 - г) хоча б одне просте число.
16. Написати вираз, значенням якого є довжина послідовності (послідовність може задаватись генератором).
- 17*. Написати вираз, що перевіряє, чи правда, що задана числова послідовність є стаціонарною.
18. Написати вираз, що задає ознаку того, що одна послідовність мажорує іншу. Довжини послідовностей вважати рівними.
 Послідовність $a_1, \dots, a_n$ мажорує послідовність $b_1, \dots, b_m$ якщо

$$n = m \wedge a_1 \geq b_1 \wedge a_2 \geq b_2 \wedge \dots \wedge a_n \geq b_n$$
19. Написати функцію **compare**, яка порівнює два булевих вектори. Якщо перший вектор менший, має повертатись **-1**, якщо вектори рівні, то **0**, якщо другий вектор менший, то **1**. Для непорівнюваних векторів має повертатися **None**. Явні цикли та рекурсію використовувати заборонено.
20. Написати інструкцію, що задає сортування списку цілих, за якого спочатку в традиційному порядку розташовуються всі його парні елементи, а потім у традиційному порядку непарні.

18 Асоціативні контейнери

Раніше розглядалися послідовнісні контейнери (кортежі, списки тощо), що зберігають послідовності даних і задають лінійне упорядкування даних, що в них зберігаються. Для перевірки входження елемента в послідовність загального вигляду в гіршому випадку буде необхідно переглянути усі елементи послідовності і використати при цьому лінійний час. Для великих обсягів даних це може бути не зовсім швидко.

Якщо відсортувати значення за зростанням, то пошук у гіршому випадку вже буде вимагати логарифмічного часу, що значно краще. Але тут постає задача: як під час виконання операцій вставки нових значень та видалення існуючих значень підтримувати дані у вигляді, придатному для швидкого пошуку, а також як не дуже довго виконувати операції вставки/видалення. Найчастіше для таких задач використовують деревоподібні структури даних.

В усіх попередніх випадках мова йшла про те, що ми працюємо з даними довільного вигляду, тобто не існує жодних обмежень на їхні значення, ми нічого не знаємо про внутрішню структуру значень і весь пошук базується виключно на порівняннях $<$, $=$.

За наявності відомостей про внутрішній устрій даних роботу з ними можна реалізувати дещо більш ефективно.

18.1 Поняття хеш-функції

Розглянемо ідею сортування підрахунками. Припустимо, що треба працювати зі значеннями в діапазоні від 0 до $255 = 2^8 - 1$. Зведемо масив на 256 елементів, в елементи якого будемо записувати кількість входжень значення в послідовність, що має бути відсортована. Далі за цим масивом можна легко записати результуючу відсортовану послідовність. У даному випадку елементи масиву є корзинами (*bucket*), в які «складаються» елементи вхідної послідовності.

Тепер припустимо, що вхідні значення знаходяться в межах від 0 до $2^{16} - 1$, але ми все одно хочемо використовувати той самий підхід. Тоді нехай вхідне значення v потрапляє в корзину, що відповідає елементу масиву з номером $v // 2^{**}8$. Тепер в елементах масиву зберігатимуться не кількості значень, а ті значення, що потрапили у відповідну корзину. Тепер вхідна задача сортування зводиться до задач сортування кожної корзини окремо, після чого знов-таки можна утворити результуючу відсортовану послідовність за вмістом відсортованих корзин. Якщо вхідні значення розподілилися по корзинах більш-менш рівномірно, то в кожену корзину потрапило небагато елементів й відсортувати їх можна відносно швидко.

Той самий підхід можна використовувати і для зберігання даних. Можна зберігати дані в невеликих за розмірами «корзинах», забезпечуючи швидкий доступ до корзини (наприклад, зберігаючи масив корзин), що відповідає значенню. А вже всередині корзини використовувати методи, що ґрунтуються на порівняннях (або ще раз розподіляти вже по інших «корзинах»).

Дещо змінимо вхідні дані. Нехай вони є рядками (довжина рядків не регламентується), що складаються з малих літер латинського алфавіту. Тоді можна, наприклад, використати 26 корзин, пронумерованих числами від 0 до 25, відправляючи рядок у корзину, що відповідає його першій літері.

Подивимось на те, що тут відбувається з точки зору математики. Нехай вхідні значення беруться з деякої множини допустимих вхідних значень *Domain*. Якщо використовується n корзин, то кожному значенню зіставляється корзина з номером, що належить множині $0, n - 1$. Тобто маємо справу з функцією $f: Domain \rightarrow 0, n - 1$. Отримана функція f , яка відображає значення з деякої предметної області в початковий відрізок натурального ряду фіксованої довжини, може бути названа **хеш-функцією** (англ. *hash function*). Основна ідея полягає, що незалежно від розміру аргументу хеш-функція повертає значення, яке може бути записане у фіксованій кількості бітів. Якщо хеш-функція записує значення в m бітах, то можна казати, що

вона повертає значення з діапазону $\overline{0, 2^m - 1}$, хоча отриману послідовність бітів можна інтерпретувати не тільки як число.

Отже, на практиці значення n , яке задає кількість корзин, по яких розподіляються дані, обирають як степінь двійки і намагаються використовувати значення кратні 2^m , де m — розрядність інтерпретатора/процесора.

Як правило, хеш-функції будують так, щоб вони швидко обчислювалися і вхідні дані більш-менш рівномірно розподілялися по корзинах. Також при побудові хеш-функції можуть враховувати статистичні особливості даних, до яких вона буде застосовуватись. Крім того, найчастіше кількість елементів множини *Domain* набагато більша за значення n . Хоча бувають задачі, в яких значення n береться рівним кількості елементів множини *Domain*. Все залежить від того, для чого планується використовувати хеш-функцію.

У мові Python наявна вбудована функція **hash**, яка обчислює *хеш-значення*, або просто *хеш*, свого аргументу. Ця функція визначена не для всіх об'єктів. В якості хеш-значень використовуються цілі числа (у діапазоні, що відповідає розрядності інтерпретатора: для 32-розрядного отримане значення буде записуватися в 32 біти, для 64-розрядного — у 64 біти; точну використовувану інтерпретатором розрядність хеш-значень можна отримати через **sys.hash_info.width**). Хеш-значення рівних значень мають бути однакові. Наприклад, ціле 1 дорівнює дійсному 1.0, тому вони мають хешуватися в однакове значення. Для вбудованих типів ця властивість виконується.

hash(1)
1

hash(1.0)
1

hash(1.0+0j)
1

print(hash(1.5), hash(1.5+0.0j), sep='\n')
1152921504606846977
1152921504606846977

Хоча хеш-значення рівних значень мають бути однакові, але ця теза має виконуватися тільки в межах одного «виклику» інтерпретатора. За різних викликів одне те саме значення може хешуватися в різні значення. Зокрема це стосується рядків.

hash('qwerty')
-6115198644443173513

За наступного виклику інтерпретатора хеш-значення повинно бути іншим.

Мова Python визначає таке поняття, як *об'єкт, який можна захешувати* (англ. *hashable object*). Це об'єкти, для яких визначено значення функції **hash**. Захешувати можна об'єкти вбудованих числових типів (у тому числі логічного), рядки типу **str**, але на цьому перелік не вичерпується. Значення, зображуване таким об'єктом, не повинно змінюватися протягом його існування. Як наслідок цей об'єкт має бути незмінюваним (англ. *immutable*). Але це більш слабка вимога. Якщо незмінюваний об'єкт є контейнером, тобто складається з частин, то й ці частини також мають хешуватись. Тобто кортеж дійсних значень захешувати можна, а кортеж списків — ні.

Якщо ми побудували власний клас, то за умовчанням об'єкти цього класу порівнюються на рівність **==** за їх ідентифікаторами і хеш-значення утворюється хешуванням ідентифікатору об'єкта.

- Рівним об'єктам мають відповідати рівні хеш-значення.

- Якщо об'єкт можна захешувати, то його значення не повинно змінюватися протягом існування цього об'єкта.

18.2 Поняття ключа

Інше важливе поняття, яке виникає в контексті організації структур даних, — ключ. Розберемося «на пальцях», що це таке.

Розглянемо дані, що зберігаються в екзаменаційних відомостях: прізвище, ім'я, по батькові студента, номер залікової книжки, отримані бали та оцінка за національною шкалою. Кожен рядок екзаменаційної відомості утворює окремий запис (англ. *record*), а його окремі частини (номер залікової книжки, оцінка за національною шкалою тощо) є його полями (англ. *field*). У межах екзаменаційних відомостей з одного предмета унікальним ідентифікатором, що визначає відповідний запис у цілому, буде номер залікової книжки. Тоді номер залікової книжки можна використати в якості **ключа** (англ. *key*) — частини запису (найчастіше одно чи кілька полів), що дозволяє однозначно знайти, тобто ідентифікувати, запис серед інших.

Наведений підхід походить з баз даних. Щодо структур даних, то там так само ключ є ідентифікатором даних. При цьому, у загальному випадку, ключ не зобов'язаний складатися з частини даних, які ідентифікує, а може, наприклад, обчислюватися за ідентифікованими даними (останнє для баз даних можливе, але не характерне).

18.3 Поняття асоціативного контейнера

Асоціативні контейнери забезпечують доступ до даних за ключем, що їх ідентифікує. При цьому цей ключ визначається даними, яким відповідає, а не їх розташуванням у структурі даних. (Порівняйте: послідовнісний контейнер задає лінійне упорядкування даних, тобто дані можна пронумерувати; але номери даних залежать від поточного вмісту контейнера, а не від значень, що їм відповідають.)

З точки зору математики можна вважати, що асоціативний контейнер зберігає пари (*ключ, значення*) і кожен ключ ідентифікує відповідну пару (*ключ, значення*), тобто даними є пари.

Щодо програмних реалізацій, то зовсім не обов'язково, що зберігаються саме пари: можуть зберігатися не тільки пари, але й тільки ключі (якщо даними є ключі, як у випадку множин) або тільки значення (якщо ключ обчислюється за значенням). Конкретика залежить від реалізації. При цьому операції пошуку, вставки, вилучення відбуваються за ключем (невелике уточнення: вставляється все-таки не тільки ключ, але й значення).

Хоча ключ і має ідентифікувати дані, тим не менш розглядають структури даних, в яких один той самий ключ відповідає кільком значенням, тобто не є унікальним. Незважаючи на парадоксальність ситуації, критичних колізій тут нема. Припустимо, що є множина з двох однакових яблук. З одного боку яблука різні, бо вони різні фізично. З іншого боку вони однакові (тобто еквівалентні, нерозрізніювані), бо мають однакові характеристики. Приблизно так само можна дивитись і на ситуацію різних значень, що відповідають одному ключу: значення можуть бути різними, але за певними ознаками еквівалентними.

Мають бути ключі унікальними чи ні визначається логікою контейнера. У випадку неунікальних ключів операція пошуку найчастіше реалізується як така, що повертає одне зі значень із заданим ключем.

Внутрішній устрій асоціативних контейнерів може бути доволі різним:

- можуть використовуватися дерева пошуку та подібні до них дерева, (потрібно мати лінійне впорядкування ключів; для ключів має бути визначено порівняння <).
- може використовуватися хешування ($\approx$ дані зберігаються в окремих корзинах), (на ключах має бути означена хеш-функція)
- може використовуватись поєднання попередніх підходів.

Асоціативні контейнери впорядкування ключів не задають, натомість, якщо на ключах задано лінійний порядок, то він може використовуватися структурою даних.

Гарно побудовані асоціативні контейнери, що використовують упорядкування ключів, у гіршому випадку виконують операції пошуку, вставки, видалення ключа швидше за асоціативні контейнери, що використовують хешування.

18.4 Асоціативні контейнери мови Python

У мові Python наявні класи **set**, **frozenset**, **dict** вбудованих асоціативних контейнерів. Усі вони підтримують унікальні ключі.

Об'єкти класу **set** моделюють множини. У випадку множини кожне значення є власним ключем і фактично зберігаються не пари (*ключ, значення*), а тільки значення. Об'єкти цього класу є змінюваними. Клас **frozenset** також моделює множини. З точки зору використання він відрізняється від класу **set** тим, що його об'єкти після створення не можуть змінювати своє значення.

Об'єкти класу **dict** (скорочення від англ. *dictionary*, словник) моделюють відображення з множини ключів у множину значень і зберігають саме пари (*ключ, значення*). Об'єкти цього класу є змінюваними.

Наведені контейнери використовують хешування ключів⁶³ (а не впорядкування), а тому елементи множин та ключі в словниках мають бути такими, що хешуються (а тому й незмінюваними). Так, елементами множин та ключами словника можуть бути цілі, дійсні числа, кортежі чисел, рядки, незмінювані множини (об'єкти типу **frozenset**) з цілих, дійсних, рядків, але ані списки, ані множини, ані словники в якості елементів множин та ключів використати неможливо.

18.5 Типові операції над множинами мови Python

Як зазначалося раніше, для зображення в програмі множин можна використати типи **set** та **frozenset**. Розглянемо більш детально роботу з об'єктами класу **set**.

Ще раз зазначимо, що:

- у випадку множини ключем є значення елемента в цілому;
- елементами множини можуть бути тільки значення, які хешуються;
- з попереднього випливає, що елементи множини мають бути незмінюваними;
- елементи множини мають бути попарно нерівними (бо ключі мають бути унікальними).

18.5.1 Створення множин

Для створення множини можна у фігурних дужках через кому записати **непорожній** перелік її елементів. Елементи множини можуть бути різних типів, але мають хешуватися. Якщо якесь значення присутнє в переліку кілька разів, то в множину воно увійде все одно рівно один раз (повторення елементів не допускаються). Більш того, порядок елементів для множини не тільки не має значення, але й навіть не зберігається в якому порядку елементи додавались у множину.

Приклади утворення об'єкта типу **set** за явним переліком елементів.

1. Значенням виразу **{1, 1, 4, 6}** буде множина **{1, 4, 6}**, бо за однаковими значеннями з переліку в множину додається тільки один елемент.

2. Значенням виразу **{1, 1.0, 4, 6}** буде множина **{1.0, 4, 6}**, бо ціле 1 та дійсне 1.0 рівні, отже з них у множині присутнє тільки одне.

⁶³ Через те, що множини та словники мови Python будуються на хешуванні ключів і не вимагають їх упорядкування, їх можна класифікувати як неупорядковані асоціативні контейнери (англ. *unordered associative containers*).

3. Наступні вирази також є синтаксично коректними та задають множини: `{(1, 1), (2, 3, 4)}` (множина, що складається з двох кортежів цілих), `{'a1', 'v3q'}` (множина з двох рядків).

4. Спроба утворити множини `{[1]}` (множина, елементом якої є список), `{([1], [2])}` (множина, елементом якої є кортеж, що складається з двох списків), буде невдалою та призведе до появи винятку типу **TypeError**, бо елементи, з яких пропонується скласти множину не є такими, що хешуються (зображуване ними значення може бути змінено).

◀

Також множини можна створювати за допомогою явного виклику конструктора. Аргументом виклику має бути об'єкт, за яким можна ітерувати, у тому числі й інша множина. Обмеження на елементи залишаються тими самими. Результуюча множина складається зі значень, що обходяться в результаті ітерування.

Приклади утворення множини за допомогою явного виклику конструктора.

Для кортежу `b = (1, 4, 5, 5, 8)` результатом `set(b)` буде така сама множина, що утвориться за виразом `{1, 4, 5, 8}`. Якщо в якості аргументу використати рядок, наприклад, `set('abcd')`, то утвориться множина, що складається з однолітерних рядків (нагадаємо: тип символів у Python відсутній, натомість символи зображуються рядками довжини 1). Вирази `set('abcd')` та `{'a', 'd', 'b', 'c'}` задають одну ту саму множину.

Щоб створити порожню множину, слід використати вираз `set()`.

◀

Для утворення множин можна використовувати конструкцію наповнювання (*comprehension*). Наприклад, за виразом `{el for el in range(5)}` буде утворено таку саму множину, що й за виразом `{0, 1, 2, 3, 4}`.

18.5.2 Обходи множин

За множинами можна ітерувати.

За виконання

```
for el in {1, 3, 5, 6, 8}:
    print(el, end=' ')
```

може бути виведено.

```
8 1 3 5 6
```

А може бути й так.

```
for el in {1, 3, 8, 6, 5}:
    print(el, end=' ')
```

```
1 3 5 6 8
```

Наведена в прикладі інструкція гарантує виконання тіла циклу для кожного елемента множини, але порядок обходу елементів **не регламентує**.

Тут важливо зазначити, що порядок обходу елементів множини у тому числі залежить від хеш-значень ключів. За різних викликів інтерпретатора можуть використовуватися різні параметри для хешування вбудованих типів (наприклад, рядків), а тому порядок обходу елементів однієї тієї самої множини може різнитися для різних викликів навіть одного того самого інтерпретатора. Вилучати та додавати елементи під час такого обходу не слід.

18.5.3 Операції, визначені над множинами

Для множин означено оператори **in** та **not in**, які перевіряють належність елемента множині. Так, значеннями виразів `1 in {1, 3}` та `1.0 in {1, 3}` є **True**. Значенням виразу `'a' in {'A', 'aa'}` є **False**. За спроби обчислити значення виразу `[1] in {1, 2}` буде згенеровано виняток типу **TypeError**, бо значення виразу `[1]` не хешується (є списком).

Якщо значенням змінної `a` є множина, то результатом виразу `len(a)` буде кількість елементів множини `a`.

```
len({1, 1, 4, 5, 5})
```

```
3
```

За обчислення істиннісного значення, результатом буде **True** тоді і тільки тоді, коли множина непорожня. Інакше кажучи, по аналогії з послідовнісними типами, значення оцінюється як істинне тоді і тільки тоді, коли функція **len()** повертає на ньому ненульове значення.

До множин застосовні оператори **==** та **!=**, що перевіряють чи є множини рівними та нерівними відповідно. Оператори **<**, **<=**, **>**, **>=** для множин відповідають теоретико-множинним порівнянням на включення **⊂**, **⊆**, **⊃**, **⊇** відповідно.

Строге включення можна перевірити, наприклад, так.

<code>{1, 3} < {1, 2, 3}</code>
True

Оператори **|**, **&**, **-**, **^** задають об'єднання, перетин, різницю та симетричну різницю множин відповідно. Також допустимі складені присвоювання **|=**, **&=**, **-=**, **^=**, які розуміються традиційно.

Наведені операції можуть бути записані не тільки через оператори, але й у вигляді викликів методів (див. табл. 30).

Табл. 30 — Синтаксичне оформлення операцій з множинами в мові Python

Вираз, записаний через оператори	Вираз, записаний через виклик методу	Примітка
<code>a<=b</code>	<code>a.issubset(b)</code>	
<code>a>=b</code>	<code>a.issuperset(b)</code>	
<code>a b1 b2 ... bn</code>	<code>a.union(b1,b2,...,bn)</code>	(*)
<code>a & b1 & b2 &...& bn</code>	<code>a.intersection(b1,b2,...,bn)</code>	(*)
<code>a - b1 - b2 -...- bn</code>	<code>a.difference(b1,b2,...,bn)</code>	(*)
<code>a^b</code>	<code>a.symmetric_difference(b)</code>	(*)
<code>a = b1 b2 ... bn</code>	<code>a.update(b1,b2,...,bn)</code>	
<code>a&= b1 & b2 &...& bn</code>	<code>a.intersection_update(b1,b2,...,bn)</code>	
<code>a-= b1 b2 ... bn</code>	<code>a.difference_update(b1,b2,...,bn)</code>	
<code>a^=b</code>	<code>a.symmetric_difference_update(b)</code>	

Зауваження.

1. Для виразів в операторній формі усі аргументи мають бути множинами. Допускається одночасне використання множин типів **set** та **frozenset**.

2. Для виразів записаних у вигляді виклику метода значення змінної **a** має бути множиною. Аргументи **b**, **b1**, **b2**, ..., **bn** виклику метода можуть не бути множинами. Достатньо, щоб по ним можна було ітерувати.

3. У рядках (*) результатом обчислення виразу є новий об'єкт того ж типу, що й **a**. Тобто тип результату визначається за першим аргументом.

4. Для незмінюваної множини **a** результатом складеного присвоювання є зв'язування імені **a** змінної з новим об'єктом, тобто інструкція **a |= b** задає те саме, що й інструкція **a = a | b** (змінна **a** зв'язується з новим об'єктом).

Приклади.

1. Об'єднання, перетин, різниця та симетрична різниця.

<code>a = {1, 2}</code> <code>b = {1, 3}</code> <code>a b</code>
<code>{1, 2, 3}</code>

<code>a & b</code>
<code>{1}</code>
<code>a - b</code>
<code>{2}</code>
<code>a ^ b</code>
<code>{2, 3}</code>

2. За використання форми запису у вигляді виклику метода тільки перший аргумент має бути множиною.

<code>a = {1, 2}</code> <code>a.union(range(3,6))</code>
<code>{1, 2, 3, 4, 5}</code>

В операторній формі обидва аргументи мають бути множинами.

<code>a range(3, 6)</code>
----- --- TypeError Traceback (most recent call last) <ipython-input-26-ed39f1479978> in <module> ----> 1 a range(3,6) TypeError: unsupported operand type(s) for : 'set' and 'range'

3. Складені присвоювання також можливі.

<code>a = {1, 2}</code> <code>b = {1, 3}</code> <code>a = b</code> <code>a</code>
<code>{1, 2, 3}</code>

Якщо **a** — множина типу **set**, то відбувається зміна об'єкта, що є значенням змінної **a**, а не утворення нового об'єкта.

Для множин можна перевіряти чи є вони диз'юнктними (тобто, чи правда, що їх перетин порожній). Відповідна перевірка записується через метод **isdisjoint**: значення виразу **a.isdisjoint(b)** є істинним тоді і тільки тоді, коли множина **a** не має спільних елементів з **b**.

<code>a = {1, 2}</code> <code>b = {1, 3}</code> <code>a.isdisjoint(b)</code>
<code>False</code>

У результаті обчислення виразу **a.add(el)** до множини **a** буде додано елемент **el**, якщо його не було; якщо був, то множина залишиться без змін. За виразами **a.remove(el)** або **a.discard(el)** з множини буде видалено елемент **el**; але в першому випадку, якщо елемент **el** у множині відсутній, то буде згенеровано виняток типу **KeyError**. Значення виразу **el** в усіх випадках має бути таким, що хешується.

Приклади.

```
a = {1, 2, 3}
a.add(4)
a
{1, 2, 3, 4}
```

```
a.remove(2)
a
{1, 3, 4}
```

```
a.remove(5)
a
-----
---
KeyError                                Traceback (most recent call
last)
<ipython-input-30-04f6dc141dde> in <module>
----> 1 a.remove(5)
      2 a

KeyError: 5
```

```
a.discard(5)
a
{1, 3, 4}
```

```
a.discard(3)
a
{1, 4}
```

Якщо елемент в множині вже є, то повторне додавання до помилки не призводить.

```
a.add(1)
a
{1, 4}
```



Якщо множина **a** непорожня, то значенням виразу **a.pop()** буде її елемент (довільний), який при цьому вилучається з множини.

```
a = {1, 3, 4}
while a:
    print(a.pop(), end=' ')
    print(a)
1 {3, 4}
3 {4}
4 set()
```

За виконання **a.clear()** з множини **a** будуть вилучені усі елементи.

```
a = {1, 3, 7}
print(a)
a.clear()
print(a)
{1, 3, 7}
```

```
set()
```

Аналогічно кортежам та спискам, у результаті виконання інструкцій

```
a = {1, 2, 3}
b = a
```

змінні **a** та **b** позначають один той самий об'єкт типу **set**, тобто надалі зміна значення змінної **a** буде тягнути такі самі зміни значення змінної **b**. Якщо для кортежів та списків для утворення неглибокої копії можна було використати зріз або метод **copy**, то для множин зрізи не визначені, але використання методу **copy** можливе.

```
c = a.copy()
```

Значенням змінної **c** є новий об'єкт множини, що містить ті самі елементи, що й множина **a**.

```
c is a
```

```
False
```

Якщо **a** позначає множину, то за виразом **set(a)** також буде утворено неглибоку копію множини **a**.

18.5.4 Незмінювані множини

Незмінювані множини, тобто множини типу **frozenset**, на відміну від множин типу **set**, є такими, що хешуються. Це досягається у тому числі через те, що об'єкт після створення не може бути змінений. З точки зору використання вони можуть використовуватися всюди, де не передбачається їх зміна і де можливо використання звичайних множин. Якщо треба задати множину, елементами якої є множини, то для елементів слід використовувати тип **frozenset**.

Для задання множини **frozenset** слід використовувати конструктор. Якщо **iterable** позначає об'єкт, за яким можна ітерувати, то тоді в результаті обчислення виразу результатом **frozenset(iterable)** буде створено множину типу **frozenset**, наповнену елементами з **iterable**. На елементи незмінюваних множин накладаються ті самі обмеження, що й на елементи звичайних множин: 1) елементи мають бути такими, що хешуються; 2) елементи незмінюваної множини є попарно різними.

Приклади.

Для створення множин **frozenset** використаємо конструктор.

```
a = frozenset({1, 3})
a
```

```
frozenset({1, 3})
```

```
b = frozenset({2, 7})
b
```

```
frozenset({2, 7})
```

```
{a, b}
```

```
{frozenset({2, 7}), frozenset({1, 3})}
```

Тип результату теоретико-множинних операцій над множинами визначається типом першого аргументу. Якщо перший аргумент множина **frozenset**, то результат також буде мати тип **frozenset**.

```
a | b
```

```
frozenset({1, 2, 3, 7})
```

```
a | {2}
frozenset({1, 2, 3})
```

Тут перший аргумент типу **set**, тому й результат також типу **set**.

```
{2} | a
{1, 2, 3}
```

Складені присвоювання до множин типу **frozenset** застосовні. Для незмінюваної множини **a** результатом складеного присвоювання є зв'язування імені **a** змінної з новим об'єктом, тобто інструкція **a |= b** задає те саме, що й інструкція **a = a|b** (змінна **a** зв'язується з новим об'єктом).

Утворимо множину **frozenset**.

```
a = frozenset({1, 3})
```

Запам'ятаємо об'єкт у додаткову змінну.

```
a_old = a
```

Виконаємо складене присвоювання.

```
a |= {7}
a
frozenset({1, 3, 7})
```

Спостерігаємо, що присвоювання виконалося. Перевіримо, чи залишився об'єкт тим самим.

```
a is a_old
False
```

Бачимо: результат записано в іншому об'єкті.

Для множин типу **set** виконання аналогічних дій призведе до зміни поточної об'єкта множини, а не до утворення нового.

```
a = {1, 3}
a_old = a
a |= {7}
a
{1, 3, 7}
```

```
a is a_old
True
```

18.6 Словники в мові Python

Словники, тобто об'єкти класу **dict**, у мові Python моделюють відображення. Як зазначалося раніше, вони зберігають пари (*ключ, значення*), де ключі є унікальними, тобто кожен ключ, що зберігається в словнику, наявний рівно в одній парі: одночасно в словнику двох різних пар з однаковими ключами бути не може. Аналогічну властивість мають відображення. Інколи навіть кажуть, що тип **dict** моделює відображення (англ. *mapping*). Якщо подивитися з точки зору математики, то значення типу **dict** задовольняє властивість функціональності; більш того можна казати, що словник задає відображення з множини ключів у множину значень.

Також нагадаємо, що ключами можуть бути тільки ті значення, які хешуються, а об'єкти-словники є змінюваними.

18.6.1 Створення словників

Для створення словника можна у фігурних дужках через кому записати перелік його пар *ключ:значення*, в якому компоненти пари відокремлюються двокрапкою. Ключі можуть бути різних типів, але мають хешуватися. На значення жодних обмежень не накладається.

Приклади.

1. Значенням виразу `{1:11, 2:21, 5:51}` буде словник з множиною значень `{(1,11), (2,21), (5,51)}` (виводиться як `{1:11, 2:21, 5:51}`).

```
{1:11, 2:21, 5:51}
```

```
{1: 11, 2: 21, 5: 51}
```

2. За виразом `{}` буде утворений порожній словник, тобто без елементів.

```
{}
```

```
{}
```

```
type({})
```

```
dict
```

3. Якщо якийсь ключ присутній у переліку кілька разів, то в словнику йому буде відповідати значення, яке згадується в переліку останнім: значенням виразу `{1:1, 1.0:5}` буде словник `{1: 5}`, бо ключі `1` та `1.0` рівні (з них вибирається перше згадування ключа), а останнім значенням для ключа є `5`.

```
{1:1, 1.0:5}
```

```
{1: 5}
```

4. Спроба використати в якості ключів значення, що не хешуються, аналогічно множинам призведе до генерації винятку типу **TypeError**.

```
{[1]:123}
```

```
-----  
---  
TypeError                                Traceback (most recent call  
last)  
<ipython-input-5-b69225cc73b6> in <module>  
----> 1 {[1]:123}  
  
TypeError: unhashable type: 'list'
```

Словники можна створювати за допомогою явного виклику конструктора. Аргументом виклику може бути інший словник: у цьому випадку створюється словник, значенням якого є неглибока копія словника-аргументу.

Приклад.

```
a = {1:11, 2:21, 5:51}
```

```
b = dict(a)
```

```
print(b)
```

```
a is b
```

```
{1: 11, 2: 21, 5: 51}
```

```
False
```

Також аргументом виклику може бути об'єкт, за яким можна ітерувати і кожне отримане значення є парою.

Приклади.

1. Утворимо словник за списком, елементами якого є двоелементні списки.

```
dict([[1,2], [3,4]])
```

```
{1: 2, 3: 4}
```

2. У випадку рядків елементами є символи, які у Python зображуються рядками довжиною 1.

```
dict(('ab', 'cd'))
```

```
{'a': 'b', 'c': 'd'}
```

3. Вираз `dict(('ab', '123'))` є помилковим, бо другий елемент складається з трьох елементів.

```
dict(('ab', '123'))
-----
---
ValueError                                Traceback (most recent call
last)
<ipython-input-9-742d871c9595> in <module>
----> 1 dict(('ab', '123'))

ValueError: dictionary update sequence element #1 has length 3; 2 is
required
```

Для утворення словників можна використовувати конструкцію наповнювання (англ. *comprehension*).

```
{x:x*x for x in range(4)}
{0: 0, 1: 1, 2: 4, 3: 9}
```

18.6.2 Обходи словників

По словнику можна ітерувати. Розглянемо приклади.

Нехай задано словник **d**.

```
d = {0:0, 2:4, 1:1, 3:4}
print(d)
{0: 0, 2: 4, 1: 1, 3: 4}
```

За інструкцією

```
for el in d:
    print(el, end=' ')
```

буде виведено.

```
0 2 1 3
```

Слід уточнити: у наведеному прикладі ітерування відбувається по ключах, що зберігаються в словнику, та в порядку, за яким ключі потрапляли в словник.

Примітка. За спроби утворити за словником список за допомогою виразу **list(d)** також буде створено список ключів словника.

```
list(d)
[0, 2, 1, 3]
```

Ітерувати можна не тільки по ключах, але й по значеннях та по парах (ключ, значення), що зберігаються в словнику. Для словника **d** значенням виразу **d.keys()** є об'єкт, який реалізує перегляд множини ключів словника; значенням виразу **d.values()** є об'єкт, що реалізує перегляд значень, що зберігаються в словнику; значенням виразу **d.items()** є об'єкт, який реалізує перегляд пар (ключ, значення), що зберігаються в словнику. Копія ані ключів, ані значень при цьому не утворюється.

По кожному з об'єктів **d.keys()**, **d.values()** та **d.items()** можна ітерувати: обхід відбувається в порядку, за яким відповідні ключі потрапляли в словник (оновлення значення ключа не враховується). У випадку **d.values()** значення може потрапляти в перелік кілька разів (якщо воно відповідає кільком ключам). Значення, що утворюються в результаті кожного з обходів об'єктів **d.keys()** та **d.items()** є унікальними, тому ці об'єкти можуть використовуватися як аргументи в теоретико-множинних порівняннях та операціях.

Продовжуємо виконувати обходи словника з попереднього прикладу.

```
for el in d.keys():  
    print(el, end=' ')
```

```
0 2 1 3
```

```
for el in d.values():  
    print(el, end=' ')
```

```
0 4 1 4
```

```
for el in d.items():  
    print(el, end=' ')
```

```
(0, 0) (2, 4) (1, 1) (3, 4)
```

Спробуємо використати **d.keys()** як множину. Обчислимо, наприклад, об'єднання.

```
{1, 2, 3} | d.keys()
```

```
{0, 1, 2, 3}
```

А тепер перетин, причому зробимо **d.keys()** першим аргументом.

```
d.keys() & {1, 2, 3}
```

```
{1, 2, 3}
```

І перевіримо строге включення.

```
d.keys() < {0, 1, 2, 3, 4, 5}
```

```
True
```

Починаючи з Python 3.8 можна здійснювати обходи словників в оберненому порядку, тобто до об'єктів, що утворюються в результаті обчислення виразів **dict.keys()**, **dict.values()** та **dict.items()**, можна застосувати **reversed**, щоб утворити ітератор, який виконує обхід в оберненому порядку. Зокрема вираз **reversed(d)** є скороченням для **reversed(d.keys())**.

```
for el in d:  
    print(el, end=' ')  
print()  
for el in reversed(d):  
    print(el, end=' ')
```

```
0 2 1 3
```

```
3 1 2 0
```

18.6.3 Доступ до елементів словника та зміна значень

Щоб отримати доступ до значення, що в словнику **d** відповідає ключу **key**, у залежності від контексту та потреб можна використати різні конструкції.

Доступ на читання. Якщо необхідно отримати значення, що відповідає ключу **key**, можна використати вираз **d[key]** або **d.get(key)**. Відмінність між цими виразами полягає в тім, що в першому випадку за відсутності ключа в словнику буде згенеровано виняток типу **KeyError**, а в другому – результатом буде значення **None**. Синтаксично **get** є методом класу, більш того він може приймати додатковий аргумент – значення, яке слід повернути, якщо ключ у словнику не знайдено. Наприклад, для словника **d = {0: 0, 2: 4, 1: 1, 3: 4}** з виклику **d.get(13, 'Ups')** буде повернуто рядок **'Ups'**.

Приклади.

```
d = {0: 0, 2: 4, 1: 1, 3: 4}
```

Знайдемо значення, що відповідає ключу 2, який наявний у словнику.

```
d[2]
4
d.get(2)
4
```

Тепер спробуємо знайти значення, що відповідає ключу, якого в словнику нема.

```
d[5]
-----
---
KeyError                                Traceback (most recent call
last)
<ipython-input-26-a63b2245b563> in <module>
      1 print(d)
----> 2 d[5]

KeyError: 5
```

Але.

```
d.get(5)
None
d.get(5) is None
True
```

Явно задамо значення, яке слід повернути за відсутності ключа. Якщо ключа нема, то повертається задане у виклику значення.

```
d.get(5, 'Ups')
'Ups'
```

Якщо ключ наявний, то повертається значення, що відповідає ключу.

```
d.get(3, 'Ups')
4
```

Доступ на встановлення значення ключа. За виконання `d[key] = value` відбудеться встановлення значення, що в словнику `d` відповідає ключу `key`. Для словника з прикладу за виконання `d[1] = 13` ключу `1` буде відповідати значення `13` (а не `1`, як було до того) і значенням словника в цілому буде `{0: 0, 2: 4, 1: 13, 3: 4}`. За спроби присвоїти значення ключу, що не існує, у словник буде додано новий ключ з відповідним значенням. Так, за виконання `d[5] = 'a'` значенням словника стане `{0: 0, 2: 4, 1: 13, 3: 4, 5: 'a'}`.

Приклад.

```
d = {0: 0, 2: 4, 1: 1, 3: 4}
```

Змінимо значення, що відповідає ключу.

```
d[1] = 13
print(d)
{0: 0, 2: 4, 1: 13, 3: 4}
```

Якщо ключа не було, то він буде доданий.

```
d[5] = 'a'
d
{0: 0, 2: 4, 1: 13, 3: 4, 5: 'a'}
```

Інколи буває необхідно додати в словник значення ключа, якщо його там нема. Цю дію можна виконати за допомогою методу **setdefault**. У результаті виконання виклику **d.setdefault(key, default)**, якщо ключа **key** в словнику нема, то до словника додається пара (**key, default**); в усіх випадках з виклику повертається значення, що відповідає ключу **key**. Аргумент **default** може бути відсутнім, тоді для нього використовується **None**, що є його значенням за умовчанням.

Приклад.

Для словника

```
d = {0:0, 2:4, 1:1, 3:4}
```

значенням виразу **d.setdefault(1, 15)** буде **1**, словник в результаті обчислення не зміниться.

```
print(d.setdefault(1,15))
d
```

```
1
{0: 0, 2: 4, 1: 1, 3: 4}
```

Значенням виразу **d.setdefault(13, 15)** буде **15**, в результаті обчислення у словник буде додано пару (13, 15).

```
print(d.setdefault(13,15))
d
```

```
15
{0: 0, 2: 4, 1: 1, 3: 4, 13: 15}
```

Якщо далі здійснити виклик **d.setdefault(10)**, то його значенням буде **None** і до словника буде додано пару (10, **None**).

```
print(d.setdefault(10))
d
```

```
None
{0: 0, 2: 4, 1: 1, 3: 4, 13: 15, 10: None}
```

◀

Вилучення ключа. За виконання інструкції **del d[key]** з словника буде вилучено ключ **key** (з математичної точки зору з множини пар, що є значенням словника буде вилучено пару (**key, d[key]**)).

Приклад (продовження).

Вилучимо ключ зі словника.

```
del d[2]
d
```

```
{0: 0, 1: 13, 3: 4, 5: 'a'}
```

Спроба вилучити ключ, що не існує, призведе до появи винятку.

```
del d[2]
d
```

```
-----
---
KeyError                                Traceback (most recent call
last)
<ipython-input-34-3b0bb3d61f18> in <module>
----> 1 del d[2]
      2 d

KeyError: 2
```

◀

Значенням виразу **d.pop(key)** буде значення, що відповідає ключу **key**, при цьому сам цей ключ буде вилючено зі словника. Виконання обчислення виразу **d.pop(key, return_if_not_found)** буде відрізнятися тільки тим, що за відсутності ключа в словнику з виклику буде повертатися значення **return_if_not_found**, натомість у попередньому випадку в цій ситуації було б згенеровано виняток типу **KeyError**.

```
d = {0:0, 2:4, 1:1, 3:4}
print(d.pop(2))
print(d)
```

```
4
{0: 0, 1: 1, 3: 4}
```

```
print(d.pop(2, 'X'))
print(d)
```

```
X
{0: 0, 1: 1, 3: 4}
```

За обчислення виразу **d.popitem()** буде повернута пара (*key, value*), яка при цьому вилючається з словника. З наявних пар завжди вибирається пара, ключ якої був доданий в словник останнім (тобто використовується принцип LIFO: **Last In First Out**). Для порожнього словника виклик призведе до появи винятку типу **KeyError**.

```
print(d.popitem())
print(d)
```

```
(3, 4)
{0: 0, 1: 1}
```

Виклик **d.clear()** робить словник **d** порожнім.

```
d.clear()
d
{}
```

Крім зазначених вище методів клас **dict** має метод **update** який виконує «комплексне» оновлення словника. Якщо в якості аргументу задати інший словник, то об'єкт словника, для якого здійснено виклик, зазнає змін:

1) будуть оновлені значення, що відповідають ключам, які наявні в словнику-аргументі, а саме їм відповідатимуть значення з словника-аргументу;

2) у словник будуть додані ключі та відповідні їм значення з словника-аргументу, які були відсутні у словнику.

```
d = {0:0, 2:4, 1:1, 3:4}
d1 = {1:2, 5:5, 13:1}
d.update(d1)
d
```

```
{0: 0, 2: 4, 1: 2, 3: 4, 5: 5, 13: 1}
```

Починаючи з версії 3.9 до словників застосовні оператори **|** та **|=**. Оператор **|=** задає оновлення словника, аналогічно методу **update**. Аргументом методу **update**, так само як і другим аргументом оператора **|=**, може бути і словник, і послідовність пар значень, яка використовується як словник.

Оператор **|** відрізняється від оператора **|=** тільки тим, що результат зберігається в новому словнику і другим аргументом має бути тільки словник.

Приклади.

```
d = {0:0, 2:4, 1:1, 3:4}
d1 = {1:2, 5:5, 13:1}
d |= d1
d
{0: 0, 2: 4, 1: 2, 3: 4, 5: 5, 13: 1}
```

```
d = {0:0, 2:4, 1:1, 3:4}
d |= [(1,2), (5,5), (13,1)]
d
{0: 0, 2: 4, 1: 2, 3: 4, 5: 5, 13: 1}
```

18.6.4 Операції, визначені над словниками

Для словників означено оператори **in** та **not in**, які перевіряють чи наявний у словнику заданий ключ. Наприклад, значенням виразу **1 in {0: 0, 1: 1}** є **True**. Значенням виразу **2 in {0: 0, 1: 1}** є **False**. Аналогічно множинам, значення яке перевіряється на входження в словник має бути таким, що хешується.

```
d = {0:0, 1:1}
2 in d
False
```

```
1.0 in d
True
```

Якщо значенням змінної **d** є словник, то результатом виразу **len(d)** буде кількість пар, що зберігаються в словнику. Словник оцінюється як істинний, тоді і тільки тоді, коли непорожній, тобто функція **len()** повертає на ньому ненульове значення.

```
d = {0:0, 1:1}
bool(d)
True
```

Для словників визначені оператори **==** та **!=**. Два словники вважаються рівними, якщо вони рівні як множини пар. Порівняння **<**, **<=**, **>**, **>=** для словників не означені.

```
{0:0, 1:1} == {1:1, 0:0}
True
```

18.6.5 Копіювання словників

Аналогічно кортежам, спискам та множинам, у результаті присвоювання словників нові об'єкти-словники не утворюються. Щоб створити новий словник за існуючим словником **d**, можна використати вираз **dict(d)** або **d.copy()**. В обох випадках відбувається неглибоке копіювання, тобто і оригінал, і його копія використовують ті самі об'єкти ключів та відповідних їм значень. У випадку ключів у цьому нема нічого поганого, бо ключі мають хешуватись, а тому бути незмінюваними. Щодо відповідних значень, то якщо вони, наприклад, є списками, то має сенс провести більш глибокий аналіз коду/алгоритму, в якому використано копіювання словника та за потреби використати глибоке копіювання за допомогою функції **deepcopy** з бібліотечного модуля **copy**.

Приклад.

```
d = {0:0, 1:1}
d1 = dict(d)
d2 = d.copy()
d.clear()
print(d1, d2, d, sep='\n')
```

```
{0: 0, 1: 1}
{0: 0, 1: 1}
{}
```

Оригінальний словник спустошено, проте копії залишилися без змін.



Проте копіювання словників є неглибоким.

Приклад.

Створимо словник **a**.

```
a = {1:[1], 2:[2]}
```

Змінній **c** присвоїмо копію словника **a**.

```
c = a.copy()
```

Зараз маємо словники **a**, **c**, значення яких однакові, але їх об'єкти різні.

Змінимо словник **a**.

```
a[3] = 3
print(f'{a=}')
print(f'{c=}')

```

```
a={1: [1], 2: [2], 3: 3}
c={1: [1], 2: [2]}
```

Змінимо значення, що зберігається в копії словника.

```
c[1].append(0)
print(f'{a=}')
print(f'{c=}')

```

```
a={1: [1, 0], 2: [2], 3: 3}
c={1: [1, 0], 2: [2]}
```

Бачимо: для обох словників змінилося зображуване ними значення. При зміні значення **c[1]** відбувається зміна значення **a[1]**, бо згідно з неглибоким копіюванням значення **c[1]** **is a[1]** істинне (ключу **1** в обох словниках відповідає один той самий об'єкт).

Продовжуємо зміни.

```
c[1] = 13
print(f'{a=}')
print(f'{c=}')

```

```
a={1: [1, 0], 2: [2], 3: 3}
c={1: 13, 2: [2]}
```

Зараз змінено не значення об'єкта, що відповідає ключу **1** у словнику **c**, а сам об'єкт словника. Така зміна на словник **a** не впливає. Після останнього присвоєння значенням виразу **c[1]** **is a[1]** буде **False**.



Приклад глибокого копіювання словника.

Утворимо словник та його глибоку копію. Аналогічно попередньому прикладу значеннями, що відповідають ключам, зробимо списки.

```
from copy import deepcopy
a = {1:[1], 2:[2]}
b = deepcopy(a)
print(f'{a=}')
print(f'{b=}')
a={1: [1], 2: [2]}
```

```
b={1: [1], 2: [2]}
```

Тепер змінимо один зі списків-значень.

```
b[1].append(0)
print(f'{a=}')
print(f'{b=}')

```

```
a={1: [1], 2: [2]}
b={1: [1, 0], 2: [2]}
```

Як можна бачити, завдяки глибокому копіюванню зображуване словником значення змінилося тільки для того словника, над яким виконувалося перетворення.



18.6.6 Ще один випадок створення словника

Бувають ситуації, коли заздалегідь відомо, які ключі мають зберігатися в словнику, але відповідні їм значення будуть встановлюватися пізніше. У цій ситуації можна утворити словник із заданою множиною ключів, яким відповідає якесь фіксоване значення. На практиці найчастіше використовується **None**. Наприклад, за виконання виклику **dict.fromkeys(range(4))** буде утворено словник **{0: None, 1: None, 2: None, 3: None}**.

Приклад. Написати функцію, що за вхідним рядком обчислює кількість входжень у нього кожної великої латинської літери.

Результат роботи функції фактично має задавати відображення з множини великих латинських літер у множину натуральних чисел. Отже, результатом функції зручно зробити словник, ключами якого є великі латинські літери, а значеннями, що відповідають ключам, – кількості входжень відповідної літери-ключа в рядок.

Якщо почати роботу з порожнього словника і додавати літери по мірі того, як вони будуть зустрічатися, то під час обробки кожної великої латинської літери буде необхідно робити перевірку, чи наявна вона в словнику: якщо її нема, то додавати ключ і встановлювати відповідне ключу значення в 1, інакше на 1 збільшувати значення лічильника входжень. Інший підхід: відразу створити лічильник для кожної літери та встановити відповідне значення в 0, а далі проходити рядок та поступово збільшувати лічильники.

Якщо планується обробка довгих рядків, то другий варіант буде більш економний за кількістю операцій. На ньому й зупинимось.

Перш за все необхідно реалізувати перелік великих латинських літер. Для цього можна використати, наприклад, такий вираз (**chr(i) for i in range(ord('A'),ord('Z')+1)**). Значенням **chr(i)** є рядок, що складається рівно з одного символу з Unicode-кодом **i**. Значенням виразу **ord('A')** є код символу **A** (функція **ord** за односимвольним рядком повертає Unicode-код цього символу). Отже, записаний вираз якраз перелічує латинські літери від **A** до **Z**. Метод **fromkeys** дозволяє другим аргументом вказувати значення, яке встановлюється для всіх ключів, що додаються в словник. Отже, початковим значенням словника **di** є словник, в якому всі лічильники встановлено в **0**.

Далі в циклі проходимо символи рядка і для кожної великої латинської літери (перевірку можна виконати, перевіривши чи наявний відповідний ключ у словнику) збільшуємо її лічильник на **1**.

Наведеному опису відповідає такий код.

```
def count_Latins(s):
    latins = (chr(i) for i in range(ord('A'), ord('Z')+1))
    di = dict.fromkeys(latins, 0)
    for a in s:
        if a in di:
            di[a] += 1
    return di

```



Зауваження з синтаксису. Метод **fromkeys** може викликатися як від імені типу **dict**, так і від імені його об'єктів. В обох випадках він створює новий об'єкт-словник; викликаний від імені об'єкта значення цього об'єкта він не змінює. У прикладі його виклик записується як **dict.fromkeys**. Під час знайомства з класами ми дізнаємося, що такі методи називають *class method*, та розберемося з ними трохи більш детально. Поки користуємося. Адже принцип абстракції даних дозволяє користуватися, не зазираючи в деталі реалізації.

18.7 Використання словників в інтерпретаторі Python

Словники використовуються інтерпретатором Python у службових цілях. Раніше мова йшла про простори імен, зокрема про те, що кожен блок коду знає простір глобальних імен (простір імен свого модуля) та має свій власний простір локальних імен. Також зазначалося, що простір імен мови Python можна розуміти як таблицю, що зіставляє імена зі зв'язаними з ними об'єктами. Зараз це можна уточнити, сказавши, що зовні простір імен виглядає як словник (насправді, використовуваний для простору імен тип не завжди буде **dict**; у деяких випадках це можуть бути дуже схожі на нього за поведінкою типи), в якому ключами є імена, а значеннями — об'єкти, що є значеннями цих імен.

Вбудована функція **globals()** повертає простір глобальних імен у вигляді словника, в якому ключами є рядки-імена, а значеннями — значення цих імен.

Вбудована функція **locals()** повертає словник з простором локальних імен блоку коду, до якого можуть додаватися вільні імена блоку (якщо блок відповідає тілу функції). Змінювати цей словник не рекомендується (у залежності від реалізації інтерпретатора це може не мати впливу на локальний простір імен або призводити до непередбачуваних наслідків). До речі, змінювати словник, отриманий з виклику **globals()**, також не краща ідея, хоч документація Python про це явно не застерігає.

Згадувана раніше в контексті імпортування системна змінна **sys.modules** також є словником, ключами в якому є імена імпортованих пакетів та модулів, а значеннями — відповідні їм об'єкти.

Надалі ми будемо розглядати організацію класів і побачимо, що члени класів (методи, атрибути тощо) також зберігаються в словниках (або схожих на них структурах).

Ще один випадок службового використання словників розглядається в наступному розділі.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: хеш-функція, **hash**, ключ, асоціативний контейнер, **set**, **frozenset**, **dict**, **copy.deepcopy**, механізм наповнювання для множин та словників.

2. У контексті множин дати стислий опис (синтаксис, семантика, особливості): **len**, **in**, **not in**, **==**, **!=**, **<**, **<=**, **>**, **>=**, **|**, **&**, **^**, **-**, **|=**, **&=**, **-=**, **^=**, **issubset**, **issuperset**, **union**, **intersection**, **difference**, **symmetric_difference**, **update**, **intersection_update**, **difference_update**, **symmetric_difference_update**, **isdisjoint**, **add**, **remove**, **discard**, **clear**, **pop**, **copy**.

3. У контексті словників дати стислий опис (синтаксис, семантика, особливості): **len**, **in**, **not in**, **==**, **!=**, **|**, **|=**, **keys**, **values**, **items**, **get**, **setdefault**, **clear**, **update**, **copy**, **[]**, **fromkeys**.

4. Для чого можуть використовуватися хеш-функції?
5. Які вимоги накладає мова Python на об'єкти, які можна захешувати?
6. Які об'єкти мови Python можна захешувати?
7. В які значення хешує бібліотечна функція **hash**?
8. Чи може залежати значення ключа від способу або місця зберігання даних, який цей ключ ідентифікує?
9. У чому полягає ключова відмінність індексів та ключів?
10. Чи можна розглядати індекси елементів послідовнісного контейнера як ключі?

11. Які структури даних задають упорядкування значень, що в них зберігаються? Які структури даних можуть використовувати при своїй побудові упорядкування значень? Чи важливе упорядкування для вбудованих типів **set**, **dict**?
12. Які з вбудованих типів мови Python є асоціативними контейнерами?
13. На яких засадах будуються типи **set** та **dict**?
14. Що зберігається в об'єктах типів **set**, **frozenset**? Які вимоги накладаються на їх вміст?
15. У чому принципова відмінність типів **set** та **frozenset**?
16. Що зберігається в об'єктах типу **dict**? Які вимоги накладаються на їх вміст?
17. Які синтаксичні конструкції дозволяють створити множину?
18. Як можна створити множину типу **frozenset**?
19. Як задати обхід множини? У якому порядку будуть обходитися її елементи?
20. Як виконуються порівняння **==**, **!=**, **<**, **<=** множин?
21. Які теоретико-множинні операції та порівняння доступні для множин у мові Python? Який синтаксис їх запису? Які допустимі типи аргументів?
22. Як визначається тип результату теоретико-множинних операцій над множинами?
23. Як додати елемент у множину?
24. Як вилучити елемент з множини?
25. Чи можливі для множин складені присвоєння? Які? Як вони записуються?
26. Які синтаксичні конструкції дозволяють створити словник?
27. Як можна задати обхід словника/його ключів/його значень/його пар? У якому порядку буде відбуватися обхід?
28. Як можна отримати значення, що в словнику відповідає ключу. Указати кілька способів. У чому відмінність між ними?
29. Як можна додати ключ у словник?
30. Як можна вилучити ключ зі словника?
31. Як можна змінити значення, що відповідає ключу в словнику?
32. Як працює оператор індексації (англ. *subscripting*) **[]** для словників?
33. Для яких вбудованих структур даних у мові Python можливо використання наповнювання (англ. *comprehension*)? Навести приклади.

Задачі та вправи

1. Написати код, який виконує такі дії.
Створити об'єкт-множину зі значенням **{2, 7}**, запам'ятати її у змінній **a**.
Перевірити, чи містить множина **a** значення **5** як елемент.
Створити об'єкт, значенням якого є порожня множина; запам'ятати її у змінній **b**.
До множини **b** додати елемент **5**.
Знайти кількості елементів множин **a** та **b**.
Перевірити, чи перетинаються множини **a** та **b**.
Перевірити, чи є **b** підмножиною множини **a**.
Для множин **a** та **b** знайти їх перетин, об'єднання, різницю та симетричну різницю.
До множини **b** додати множину **a**.
Знайти кількість елементів множини **b**.
З множини **b** вилучити елемент **7**.
Після кожної дії виводимо результат її виконання (або явно, або доручаємо це інтерпретатору).
2. Написати вираз, що задає множину, що складається з послідовних цілих чисел від 2 до 23 включно.
3. Написати вираз, що задає множину, що складається з квадратів послідовних цілих чисел від 1 до 13 включно.
4. Написати вираз, що задає множину, що складається з квадратів чисел, що містяться в послідовності **s**.

5. Задано функцію f дійсного аргументу та множину a , що складається з дійсних чисел. Написати вираз, що задає множину $f(a)$.

6. Написати вираз, що задає множину, що складається з натуральних чисел, що не перевищують 100 000 та не діляться на жодне з чисел 2, 3, 7.

7. Задано множини a та b , що складаються з цілих чисел. Знайти множину, що складається з тих і тільки тих елементів множини a , що не діляться на жодне число з множини b . Розв'язок оформити у вигляді функції (але тіло функції має складатись рівно з однієї інструкції **return**).

8. Написати функцію, що повертає кількість різних дійсних значень, що містяться в послідовності s .

9. За послідовністю цілих чисел s визначити яких значень в ній більше: парних чи непарних. Кожне значення враховується рівно один раз. Якщо більше парних, повернути 2, якщо більше непарних повернути 1, якщо кількості однакові, повернути 0. Розв'язок оформити у вигляді функції.

10. Написати функцію, що виводить елементи множини s за зростанням.

11. Написати код, який виконує такі дії.

Створити об'єкт-словник зі значенням $\{1:2, 0:3\}$, запам'ятати в змінній a .

Додати до словника пару (3,6).

Вивести словник a , його ключі, його значення та пари, що в ньому зберігаються.

Збільшити значення, що відповідає ключу 0, на 5.

Вивести вміст словника (пари ключ:значення) за зростанням ключів.

Перевірити, чи наявний у словнику ключ 13.

Вилучити зі словника ключ 0.

Знайти кількість ключів у словнику.

Після кожної дії виводимо результат її виконання (або явно, або доручаємо це інтерпретатору).

12. Написати вираз, значенням якого є словник, в якому ключами є цілі від 0 до 10, а значеннями – їх квадрати.

13. Написати вираз, значенням якого є словник, в якому ключами є цілі від 0 до 100, а всі значення дорівнюють 0.

14. Написати функцію, що за цілим n знаходить кількість входжень кожної десяткової цифри в його запис. Результат обчислення має бути словником.

15. Написати функцію, що вилучає зі словника усі ключі, яким відповідають нулеві значення.

16. Написати функцію, що в словнику всі парні значення збільшує на 1.

17. Написати функцію, що за цілим n знаходить його канонічний розклад на прості множники (у вигляді словника з парами *простий множник: кількість його входжень*).

18. Словник a містить канонічний розклад натурального числа n на прості множники. Написати функцію, що за словником a обчислює число n .

19. Словники $a1$ та $a2$ містять канонічні розклади натуральних чисел $n1$ та $n2$ на прості множники. Написати функцію, що знаходить:

а) канонічний розклад найменшого спільного кратного цих чисел;

б) канонічний розклад найбільшого спільного дільника цих чисел;

в) найменше спільне кратне цих чисел;

г) найбільший спільний дільник цих чисел.

20. Написати функцію, що за послідовністю цілих, утворює словник, що зберігає множину її елементів як множину з повтореннями.

21. Написати функцію, що за словником, що зберігає множину з повтореннями, утворює список, що зберігає ту саму множину (елемент входить у список стільки разів, скільки разів він входить у множину).

22. Написати функцію, що за словником, що зберігає множину з повтореннями, знаходить кількість її елементів.

23. Написати функцію, що за словником обчислює скільки разів зустрічається в ньому задане значення.

24. Написати функцію, що за послідовністю цілих буде список її елементів, що зустрічаються в ній рівно один раз.

25. Написати функцію, що за послідовністю чисел буде список її елементів, що зустрічаються в ній рівно один раз. Числа, що мають різний тип, вважаємо різними.

26. Написати функцію, що за послідовністю цілих буде список її елементів, що зустрічаються в ній більше одного разу.

27. Написати функцію, що за послідовністю цілих знаходить елементи, що зустрічаються в ній найчастіше.

28. Написати функцію, що за послідовністю цілих знаходить 5 елементів, що зустрічаються в ній найчастіше. Якщо п'яте місце розділяють кілька значень, додавати до результату всіх їх.

29. Написати функцію, що за послідовністю цілих знаходить елементи, що зустрічаються в ній рідше за інші (але зустрічаються).

30. Написати функцію, що за послідовністю цілих знаходить всі елементи, що зустрічаються в ній рівно 7 разів.

31. Написати функцію, що перевіряє чи складаються дві послідовності з однакових елементів. Кількість входжень елемента в послідовність до уваги не брати. Порядок елементів у послідовності несуттєвий. Наприклад, послідовності (1, 6, 1, 3) та (3, 1, 6, 6) складаються з однакових елементів.

32. Написати функцію, що перевіряє чи складаються дві послідовності з однакових елементів з точністю до кількості входжень кожного елемента в послідовність. Порядок елементів у послідовності несуттєвий.

33. Список містить цілі числа. Вилучити зі списку усі повторення (кожне значення має залишитися в одному екземплярі). Для кожного числа має залишатись його перше входження в список.

34. Список містить числа. Вилучити зі списку усі повторення (кожне значення має залишитися в одному екземплярі). Для кожного числа має залишатись його перше входження в список. Числа, що мають різний тип, вважаємо різними.

35. Написати функцію, що за функцією f типу $Z \rightarrow Z$ та множиною A знаходить множину $f(A)$. Вважати, що функція задана словником, а множина A — множиною. Результат повернути у вигляді множини.

36. Для функції f типу $A \rightarrow B$ обернена функція, узагалі-то кажучи, функцією з $B \rightarrow A$ може не бути. Тим не менш можна розглядати "обернену" функцію g з B в $P(A)$ (булеан множини A), $g(b) = f^{-1}(b)$ ($f^{-1}(b)$ — повний прообраз елемента b). Написати функцію, що за функцією f типу $Z \rightarrow Z$, знаходить "обернену" до неї функцію g як функцію з Z в $P(Z)$. Вважати, що функцію f задано словником, результатом функції також має бути словник.

37. За функцією f , заданою словником, та елементом b знайти значення $f^{-1}(b)$. Розв'язання оформити у вигляді функції, що приймає f та b як аргументи.

38. За функціями f та g , заданими словниками знайти їх композицію. Розв'язання оформити у вигляді функції, що приймає словники f та g як аргументи та результат повертає у вигляді словника.

Множина з повтореннями допускає, що множина може містити деякі елементи більше одного разу, наприклад $\{1, 3, 3, 4\}$. Множину з повтореннями можна розглядати як відображення з множини її елементів у множину додатних натуральних чисел: кожному елементу відповідає кількість його входжень. Тоді множину $\{1, 3, 3, 4\}$ математично можна зобразити як відповідність $\{(1,1), (3,2), (4,1)\}$ між множинами $\{1, 3, 4\}$ та N_+ .

39. Для множин з повтореннями, заданими словниками, реалізувати теоретико-множинну операцію перевірки

а) строгого включення; б) включення.

Розв'язання оформити у вигляді функції.

40. Для множин з повтореннями, заданими словниками, реалізувати теоретико-множинні операції:

а) об'єднання; б) злиття; в) перетин; г) різниця.

Розв'язання оформити у вигляді функції.

41. Відношення C між множинами A та B ($A, B \subset Z$) задано словником, в якому ключами є елементи множини A , а значенням кожного ключа a — його повний образ $C(a) = \{b | (a, b) \in C\}$. Реалізувати функції, що перевіряють:

а) чи є відношення C функціональним; б) чи є відношення C ін'єктивним.

42. Відношення C між множинами A та B ($A, B \subset Z$) задано списком пар. Реалізувати функції, що перевіряють:

а) чи є відношення C функціональним; б) чи є відношення C ін'єктивним.

43. Функція f між множинами A та B (елементи множини A є такими, що хешуються) задана списком пар. Написати функцію, що зображує f у вигляді словника, де значеннями ключів є значення функції.

44. Відповідність C між множинами A та B ($A \subset Z$) задана списком пар. Написати функцію, що зображує C у вигляді словника, де значеннями ключів є їх повні образи.

45. Функція f між множинами A та B (елементи множини A є такими, що хешуються) задана словником, в якому значеннями ключів є значення функції. Написати функцію, що зображує f у вигляді списку пар.

46. Відповідність C між множинами A та B ($A \subset Z$) задана словником, в якому значеннями ключів є їх повні образи. Написати функцію, що зображує C у вигляді списку пар.

Бінарне відношення на множині $A = \overline{0, n-1}$ можна задавати принаймні трьома способами:

1) матрицею відношення (значення цілі);

2) матрицею відношення (значення логічного типу);

3) множиною (списком) пар.

Зараз розглядатимуться відношення, задані списком пар.

47. Бінарне відношення на множині $\overline{0, n-1}$ задане множиною пар.

Перевірити, чи є відношення:

а) рефлексивним;

г) антисиметричним;

б) антирефлексивним;

д) транзитивним.

в) симетричним;

48. Бінарне відношення на множині $\overline{0, n-1}$ задане множиною пар. Побудувати обернене відношення.

49. Розбиття множини $\overline{0, n-1}$ можна задавати принаймні чотирма способами:

1) Розбиття задається як множина, елементами якої є множини.

Примітка. У задачах, де треба будувати розбиття, використовувати множини множин зовсім незручно, адже елементи-множини мають бути незмінюваними. Насправді, в алгоритмах, де виникає потреба в розбиттях чи необхідність їх побудови, виявляється, що для збереження елементів розбиття достатньо використовувати більш просту структуру даних — список, елементами якого є множини.

2) Розбиття задається як список, елементами якого є множини.

3-4) Розбиття задається списком, що задає належність елементів класам. Окремо зберігаються елементи-множини розбиття. Класи можна пронумерувати (3) або позначати самим класом (4).

Побудувати функції, що за одним зображенням розбиття будують інше.

По наступних задачах слід урахувати всі можливі способи зображення розбиттів (див. задачу 49).

50. За еквівалентністю на множині $\overline{0, n-1}$ побудувати її фактор-множину.

51. За еквівалентністю на множині $A = \overline{0, n-1}$ та елементом множини A знайти його клас еквівалентності.

52. За розбиттям множини побудувати відповідну еквівалентність.

53. Визначити, чи є одне розбиття підрозбиттям іншого.

54. За відображенням f , заданим на множині $A = \overline{0, n-1}$, побудувати еквівалентність R_f , де

$$xR_f y \Leftrightarrow f(x) = f(y)$$

55. За розбиттям множини $A = \overline{0, n-1}$ та відображенням $f: A \rightarrow \text{bool}$ побудувати підрозбиття таке, що два елементи знаходяться в одному класі підрозбиття тоді і тільки тоді, коли відображення f приймає на них однакове значення.

56. За розбиттям множини $A = \overline{0, n-1}$ та відображенням f , заданим на множині A , побудувати підрозбиття таке, що два елементи знаходяться в одному класі підрозбиття тоді і тільки тоді, коли відображення f приймає на них однакове значення.

Сімейство множин це множина, елементами якої є множини. Надалі будемо вважати, що сімейство множин задається як множина, елементами якої є незмінювані множини.

57. Написати функцію, що за сімейством множин перевіряє, чи є елементи сімейства диз'юнктними.

58. Написати функцію, що обчислює об'єднання сімейства множин. Об'єднання порожнього сімейства вважати порожньою множиною.

59. Написати функцію, що обчислює перетин сімейства множин. За спроби обчислити перетин порожнього сімейства має генеруватись виняток.

60. Написати функцію, що перевіряє, чи є задане сімейство множин розбиттям заданої множини A .

61. Написати функцію, що перевіряє, чи є задане сімейство множин розбиттям заданої множини $\overline{0, n-1}$.

62. Написати функцію, що перевіряє, чи є задане сімейство множин покриттям заданої множини A .

63. Написати функцію, що перевіряє, чи є задане сімейство множин покриттям заданої множини $\overline{0, n-1}$.

64. **Консольне меню.** На вхід з командного рядка отримується ціле додатне число. Під час роботи програми користувач може виконувати над ним певні дії, які можуть виконуватися успішно (тоді поточне значення змінюється) або ні (тоді поточне значення залишається без змін).

Підтримуються такі дії:

- збільшити значення в 3 рази та додати 1; дія виконується успішно тільки для непарних чисел;

- поділити поточне число націло на 2; дія виконується успішно тільки для парних чисел;

- перейти до числа, що отримується, якщо поточне число записати тими самими цифрами, але в оберненому порядку; дія виконується успішно, якщо число не ділиться націло на 10.

Програма в циклі запитує в користувача бажану дію та виконує її. Мають підтримуватися чотири команди: три команди відповідають діям над значенням та ще команда **stop**, за якою припиняється цикл. Якщо задано некоректну команду, то виводиться відповідне повідомлення. Якщо дія виконана неуспішно, то виводиться відповідне повідомлення. Робота завершується тільки за командою **stop**.

Перед введенням поточної команди користувач повинен бачити поточне число, кількість успішно виконаних дій кожного виду та довідку по командам меню.

Після завершення циклу має бути виведено повідомлення про те, що роботу завершено, поточне значення числа та кількість успішно виконаних дій кожного виду.

19 Функції, що здатні обробляти змінну кількість аргументів

Під час розгляду функцій йшла мова про те, що не можна означити дві різні функції, що мають однакове ім'я. Тим не менш, серед бібліотечних функцій є багато таких, що можуть використовуватись з різною кількістю та/або з різними наборами аргументів виклику. Одним з цих механізмів є використання для аргументів значень за умовчанням, але він може допомогти не в усіх ситуаціях. Інший, набагато більш потужний механізм, розглянемо зараз.

19.1 Функції зі змінною кількістю позиційних аргументів виклику

Однією з найбільш часто вживаних функцій, що вміє працювати зі змінною кількістю позиційних аргументів виклику, є функція **print**, що виводить свої позиційні аргументи.

Напишемо функцію **print_id**, що виводить ідентифікатори (англ. *identity*) своїх позиційних аргументів по одному значенню на рядок. Синтаксично це можна оформити так.

```
def print_id(*args):  
    for arg in args:  
        print(id(arg))
```

У випадку, коли функція повинна обробляти змінну кількість позиційних аргументів, у переліку формальних параметрів зазначається формальний параметр, якому передують знак *. Семантика цього така. Під час виклику **print_id(1, 2)** аргументи **1** та **2** пакуються в кортеж **(1, 2)** і цей кортеж стає значенням **args** під час виклику. Як можна бачити, у тілі функції змінна **args** обробляється як послідовність.

Параметри, записані після параметра зі знаком *, можуть передаватися тільки як ключові. Параметри, записані перед параметром зі знаком *, мають передаватися тільки як позиційні.

У випадку функції

```
def f(a,b,*args):  
    print(a)  
    print(b)  
    print(args)
```

та виклику **f(1, 2, 3, 4)** «зайві» позиційні аргументи 3, 4 будуть запаковані в кортеж **(3, 4)** який буде передано в якості значення параметра **args**. За виконання цього виклику буде виведено таке.

```
1  
2  
(3, 4)
```

Якщо знак * використати під час запису переліку аргументів виклику, то він буде мати протилежний ефект. Нехай виконано інструкцію **a = (1, 2)**. Виклик **f(a)** є синтаксично некоректним, оскільки функція **f** має приймати два аргументи, а у виклику наявний тільки один. Натомість, можна записати **f(*a)** і цей запис буде еквівалентний **f(1, 2)**. У даному випадку знак * задає розпакування послідовності в окремі аргументи. Якщо кількість елементів не буде відповідати необхідній кількості аргументів виклику, то виклик також буде помилковим. Розпакування можливе не тільки для кортежів, але й для списків та інших об'єктів, по яких можна ітерувати.

Приклади.

```
a = (1, 2)  
a  
print(a)  
(1, 2)  
print(*a)  
1 2
```

```
f(a)
-----
---
TypeError                                Traceback (most recent call
last)
<ipython-input-9-730f2a0fa0b7> in <module>
----> 1 f(a)

TypeError: f() missing 1 required positional argument: 'b'
```

```
f(*a)
1
2
()
```

Проте, коли відбувається присвоювання списку змінних, розпакування явно записувати не треба.

```
x, y = a
print(x, y)
1 2
```

```
b = [3, 4]
print(b)
[3, 4]
print(*b)
3 4
s = {1, 6, 7}
print(*s)
1 6 7
s = {1:11, 6:11, 7:11}
print(*s)
1 6 7
```

19.2 Функції зі змінною кількістю ключових аргументів виклику

Щойно було написано функцію, що здатна обробляти довільну кількість позиційних аргументів. Зараз напишемо функцію, що може обробити довільну кількість ключових аргументів. Теоретично, якщо можливі імена ключових аргументів заздалегідь зафіксовані, то проблему можна вирішити з використанням значень за умовчанням. Але є бажання написати функцію, що вміє обробляти не просто довільну кількість ключових аргументів, а ще й з майже довільними іменами.

У випадку позиційних аргументів (їх можна розглядати як «анонімні») вони ідентифікувалися виключно своєю позицією і природним було запакувати їх у кортеж. Ключові ж аргументи зіставляють імені параметра (ключу) його значення. Отже тут є природним використання словника. Саме так це і робиться в мові Python.

Напишемо функцію, що виводить перелік значень своїх ключових аргументів.

```
def kwargs_print(**kwargs):
    for key, value in kwargs.items():
        print(key, '=', value)
```

Якщо формальному параметру в заголовку функції передують знак ******, то він розглядається як словник. Під час виклику **kwargs_print(arg1=13, arg2='34')** ключові аргументи пакуються в словник **{'arg1': 13, 'arg2': '34'}**, який під час виклику стає значенням параметра **kwargs**. За наведеного виклику буде виведено.

```
arg1 = 13
arg2 = 34
```

По аналогії із позиційними аргументами в словник пакуються тільки відсутні у заголовку функції ключові аргументи.

Функція одночасно може мати змінну кількість і позиційних параметрів, і ключових. У цьому випадку в переліку формальних параметрів параметр з позначкою ****** має вказуватися останнім.

Приклад функції, що здатна обробляти змінну кількість і позиційних, і ключових аргументів.

```
def f1(a,b,*args, c, d, **kwargs):
    print(f'{a=}\n{b=}\n{c=}\n{d=}')
    print(f'{args=}\n{kwargs=}')
```

Наведена функція виводить свої аргументи.

```
f1(1, 2, 3, 4, 5, c='q', d='w', smth='???')
a=1
b=2
c='q'
d='w'
args=(3, 4, 5)
kwargs={'smth': '???'}
```

Проте виклик

```
f1(1, 2, 3, 4, 5, c='q', d='w', smth='???', a=13)
```

призведе до появи винятку, адже формальний параметр **a**, записаний перед ***args** в означенні функції **f1**, має передаватися виключно як позиційний і спроба передати його як ключовий розцінюється інтерпретатором як спроба двічі підставити значення на місце формального параметра **a**.

Проте ситуація, коли ключовий аргумент має розглядатись не як позиційний, що був задекларований в заголовку функцій, а саме як ключовий, що відноситься до змінного переліку ключових аргументів, не така вже і рідкість. З ситуації можна вийти, якщо явно повідомити інтерпретатору, що відповідний позиційний аргумент є виключно позиційним. У такому випадку його ім'я може позначати ключовий аргумент у змінному переліку ключових аргументів. Синтаксично це оформлюється з використанням роздільника **/**.

Приклад. Для функції

```
def f2(a, b, /, *args, c, d, **kwargs):
    print(f'{a=}\n{b=}\n{c=}\n{d=}')
    print(f'{args=}\n{kwargs=}')
```

під час виклику ім'я **a** може використовуватись як ключ. Тому такий виклик є коректним.

```
f2(1, 2, 3, 4, 5, c='q', d='w', smth='???', a=13)
a=1
b=2
c='q'
d='w'
args=(3, 4, 5)
kwargs={'smth': '???' , 'a': 13}
```

Ключовий аргумент виклику **a** потрапив у перелік ключових аргументів і це не спричинило колізій з іменем тільки позиційного параметра **a**.

Аналогічно *, знак ** у переліку аргументів виклику задає розпакування словника в перелік ключових аргументів.

Приклад.

```
def f3(x, y):  
    print(x, y, sep=', ')
```

```
d = {'x':1.5, 'y':23}  
f3(**d)
```

```
1.5, 23
```

19.3 Використання функцій зі змінною кількістю аргументів

У першу чергу функції за змінною кількістю аргументів використовуються тоді, коли треба обробляти різні набори аргументів виклику, але синтаксис чітко визначає ім'я функції і використовувати для цих потреб кілька функцій з різними іменами синтаксично неможливо. Один з таких випадків виникає під час створення об'єктів.

Функції зі змінною кількістю позиційних аргументів можуть бути доцільні у випадках, коли треба однотипно обробляти одне-два значення і більший перелік, або кількість оброблюваних значень заздалегідь невідома. Якщо написати функцію, що обробляє послідовність значень, то для невеликої кількості значень може бути обтяжливим самотійно пакувати їх у викликах у кортеж. Варіант з кількома різними функціями для різних випадків призведе або до дублювання коду, або до появи зайвих функцій-обгортки з дуже схожими назвами. Отже, одна функція зі змінною кількістю аргументів тут може бути найкращим виходом з ситуації.

Функції зі змінною кількістю ключових аргументів можуть бути доцільні у випадках, коли функція може приймати багато різних налаштувань у різних комбінаціях і/або цей перелік може згодом змінюватися. Тоді замість того, щоб вводити купу окремих аргументів, краще обробляти саме словник.

Коли кількість різноманітних налаштувань програмних компонентів стає значною і оброблення можливих комбінацій вже потребує складної логіки, то однією з розповсюджених рекомендацій (шаблоном проєктування) є замість використання купи окремих параметрів ввести мову налаштувань, тобто передавати налаштування не окремими значеннями, а рядком, що містить назви параметрів та їх значення. У випадку мови Python, якщо йдеться про взаємодію коду, з обох боків написаного мовою Python, замість рядка (для якого потім буде необхідно виконувати лексичний розбір) більш доречно передавати словник з параметрів та їх значень, що можна здійснити з використанням обробки змінної кількості ключових аргументів.

Для випадків, коли в залежності від кількості аргументів, їх змісту (назв) та їх типів функція має виконувати різні дії, теж можна використати функцію зі змінною кількістю аргументів, як уніфіковану обгортку окремих функцій, що обробляють певні ситуації. (Такий підхід, наприклад, використовують для функцій, що виконують створення об'єктів.) Але зловживати цим не варто, бо за однаковість інтерфейсу прийдеться заплатити часом виконання умовної логіки, що здійснює відповідні розгалуження обробки.

Крім того «універсальність», що намагається поєднати доволі різні речі, не вважається гарним стилем у сучасному програмуванні, бо насправді ускладнює розроблення. У багатьох випадках краще мати кілька дрібних функцій, кожна з яких обробляє конкретну ситуацію, ніж одну універсальну, що здатна обробити абсолютно все.

Контрольні запитання

1. Чи можливі в мові Python функції зі змінною кількістю позиційних аргументів?
2. Чи можливі в мові Python функції зі змінною кількістю ключових аргументів?
3. Як у тілі функції зі змінною кількістю аргументів можна отримати доступ до незадекларованих явно, але наявних аргументів виклику?
4. В яких випадках має сенс використання функцій зі змінною кількістю позиційних аргументів? Наведіть приклад вбудованої функції зі змінною кількістю позиційних аргументів.
5. В яких випадках має сенс використання функцій зі змінною кількістю ключових аргументів?
6. Яку семантику має знак * у заголовку та виклику функції?
7. Яку семантику має знак ** у заголовку та виклику функції?

Задачі та вправи

1. Написати функцію, яка може приймати скільки завгодно позиційних аргументів та повертає кількість отриманих аргументів.
2. Написати функцію, яка за наявності ключового аргументу **shevchenko** виводить кілька рядків з віршів Т.Г.Шевченка, за наявності ключового аргументу **algebra** виводить формулювання основної теореми алгебри, за наявності ключового аргументу **dm** виводить означення симетричного відношення.
3. Написати функцію, що утворює множину з повтореннями (див. задачі до попереднього розділу) за своїми аргументами. Передбачити, що аргументами можуть бути і окремі елементи, і контейнери.

Приклади викликів:

```
multiset(1,1,3,3,5)
multiset(source=(1,1,1,2,2, 3, 5, 5, 6,7,7))
multiset(1,5, source=(1,1,1,2,2, 3, 5, 5, 6,7,7))
```

20 Поняття складності обчислень

20.1 Швидкодія (performance) та її аналіз

Припустимо, що в нас є програма, яка керує автопілотом (отримує поточні показники приладів і корегує курс, швидкість тощо). Зрозуміло, що дуже актуально, щоб вона своєчасно виконувала розрахунки. При обробці менш важливих для життя людини даних теж бажано отримувати відповідь не через тисячу років. Для деяких задач питання про те скільки кави встигнемо випити поки виконуються розрахунки теж має значення. Отже *час роботи* є важливою характеристикою програмного забезпечення.

Інша важлива характеристика — витрата ресурсів, наприклад, оперативної пам'яті. Для розподілених систем (складаються з кількох обчислюваних вузлів, з'єднаних у мережу) важливою характеристикою є трафік (кількість інформації, що передається по мережі за одиницю часу), бо якщо трафік перевищує пропускну здатність мережі, то виконання програми буде гальмуватись. Загальна кількість інформації, особливо допоміжної, що передається по мережі також важлива, оскільки передача інформації по мережі не тільки вимагає ресурсів, але виконується відносно повільно.

З цієї точки зору програми, алгоритми, задачі аналізують на використання часу та пам'яті. Щодо потоків інформації, цей аналіз має значення для специфічного кола задач і зараз розглядатись не буде.

Час роботи характеризується *часовою складністю* (*time complexity*), кількість необхідної для функціонування пам'яті характеризується *просторовою*, або, *ємнісною складністю* (*space complexity*). Розглянемо більш детально часову складність.

20.2 Емпіричний підхід

Нехай в нас є якась програма. Як виміряти час її роботи? Зрозуміло, що він залежить від вхідних даних.

Припустимо, що в нас є набір вхідних даних D_1 . Тоді запустимо програму над цим набором даних та виміряємо час секундоміром. А на наборі даних D_2 ? Теж запустимо і виміряємо. Але на всіх можливих даних виміряти не вдасться. Як все-таки оцінити час роботи на вхідних даних D ?

Перш за все зазначимо, що *час роботи* розглядають як *функцію від розміру вхідних даних*.

Як вимірювати розмір? У залежності від задачі в програмуванні⁶⁴ він може бути

а) кількістю елементів базових типів на вході алгоритму (найчастіше);

наприклад, якщо треба знайти найменший елемент заданої послідовності.

б) кількістю бітів, необхідних для запису вхідних даних;

наприклад, кількістю бітів числа в так званій «довгій» арифметиці, коли числа зображуються не стандартними типами, а значно довгими послідовностями цифр;

в) парою, трійкою чисел тощо;

наприклад, парою (n, m) розмірностей матриці (але для деяких задач обробки матриць розміром задачі може бути й кількість їх елементів).

Тоді можна знайти час роботи $T(n)$ для даних розміру $n = 1, 2, \dots, K$. Але не обов'язково проводити підрахунки для всіх поспіль значень від 1 до K ; можливо виконати їх тільки для деяких, наприклад: 100, 250, 1000, 2000, 5000, ... І далі спробувати визначити $T(n)$ як функцію від n . При цьому завжди вважається, що час роботи насправді є не $T(n)$, а $\lceil T(n) \rceil$ (найменше ціле, що не менше $T(n)$; $\lceil 10 \rceil = 10$, $\lceil 4.5 \rceil = 5$).

⁶⁴ У теорії алгоритмів під розміром задачі розуміється виключно кількість символів зображення вхідних даних.

При аналізі алгоритмів найчастіше розглядають функції вигляду a , $a \log n$, $an + b$, $an \log n$, an^2 , an^3 , an^b , 2^n , $a \cdot b^n$.

Методом вимірювань будуємо графік залежності $T(n)$ від n і намагаємося підібрати функцію, яка йому найбільш відповідає. Графічне зображення може допомогти визначити клас функцій, серед яких варто підбирати функцію.

У випадку пошуку функції вигляду an^b можна розглядати значення n 1, 2, 4, ..., 2^K та будувати log-log графік (графік залежності $\log T(n)$ від $\log n$).

Хоча для такого методу оцінки часу не треба знати внутрішній устрій програми та реалізовані алгоритми, але він не дає жодних гарантій і суттєво залежить від використаного апаратного та програмного забезпечення. Наприклад, від процесора, на якому виконується програма; від операційної системи, що керує комп'ютером. У випадку, коли час оцінюється не за виконуваним файлом, а за кодом, чи у випадку інтерпретованих програм, час також буде залежати від використаного компілятора/інтерпретатора.

20.3 Математичний підхід

Математичний підхід до оцінювання часу формулюється так: за програмою знайти кількість операцій кожного можливого типу і порахувати загальний час роботи.

Нехай можливі операції $op_1, op_2, \dots, op_k$, кожна з яких вимагає $c_1, c_2, \dots, c_k$ одиниць часу і використовується в програмі $n_1, n_2, \dots, n_k$ разів, тоді

$$T(n) = c_1 n_1 + c_2 n_2 + \dots + c_k n_k.$$

Така модель насправді буде відображати «чистий» час виконання коду програми, нехтуючи при цьому втратами часу, пов'язаними із сторінковою організацією пам'яті та розподілом ресурсів (зокрема, процесора) між різними процесами.

Хоча сучасні процесори трохи швидші за процесори 10-річної давнини, проте співвідношення часу обчислення різних операцій (див. табл. 31) насправді зберігається.

Табл. 31 — Час виконання арифметичних операцій показано в таблиці
(дані взято з інтернету, 2014 рік)

операція	час виконання в наносекундах
int +	2.1
int *	2.4
int /	5.4
float +	4.6
float *	4.2
float /	13.5
sin	91.3
arctg	129.0

Крім арифметичних операцій також рахуються операції:

- означення скалярної змінної,
- присвоювання,
- цілочислового та дійсного порівняння,
- організація виклику функції (виконання тіла не входить).

Для спрощення вважають, що виконання кожної з них займає фіксований константний час.

За реальних обчислень доволі часто вигляд виразу для $T(n)$ суттєво спрощують, наприклад, вважають, що всі операції виконуються однаково кількість часу.

Дуже поширеним є підхід, коли час рахують не у фізичних одиницях часу, а в якості одиниці часу беруть 1 елементарну операцію (присвоювання, порівняння, додавання тощо). Це дозволяє розглядати часову складність не тільки виконуваного коду чи програм, але й час алгоритмів.

Також інколи нехтують доданками менших степенів (і, навіть, як побачимо далі, константами), тобто розглядають *асимптотичні оцінки складності*. Наприклад, $16n^3 + n - 4 \sim 16n^3$. Якщо $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$, то кажуть, що функції $f(n)$ і $g(n)$ мають однаковий порядок зростання і позначають це так $f(n) \sim g(n)$.

20.4 Час найгіршого випадку та час у середньому

Розглянемо приклад⁶⁵: знайти заданий елемент **sample** у послідовності **a**, повернути позицію першого входження елемента в послідовність або -1, якщо елемент не знайдено. Скористаємося таким алгоритмом: будемо переглядати елементи послідовності, поки не дійдемо до кінця або поки не знайдемо шуканий елемент.

Для запису алгоритму використаємо такі домовленості та позначення⁶⁶. Розглядаємо послідовності, індексація елементів яких починається з 0. Нехай **a** позначає послідовність, тоді **len(a)** позначає довжину послідовності **a**. Для цілого **i**, $0 \leq i < \text{len}(\mathbf{a})$, **a[i]** позначає елемент послідовності з індексом **i** (**i**-й елемент).

```
def linear_search(a, sample):
    i = 0
    n = len(a)
    while i < n and a[i] != sample:
        i += 1
    if i < n:
        return i
    else:
        return -1
```

Використовувані позначення. Розглядаємо послідовності, індексація елементів яких починається з 0. Нехай **a** позначає послідовність, тоді **len(a)** позначає довжину послідовності **a**. Для цілого **i**, $0 \leq i < \text{len}(\mathbf{a})$, **a[i]** позначає елемент послідовності з індексом **i** (**i**-й елемент).

Нехай пошук виконується в послідовності 2, 3, 5, 7, -4, 8. Якщо шукати 5, то умова циклу перевірятиметься три рази (і тіло виконується 2 рази), якщо шукати 8, то 6, а при пошуку 2 — тільки один раз. Як бачимо, час виконання різний і він залежить не тільки від довжини вхідних даних (довжини послідовності), а в тому числі від того елемента, який шукаємо. З огляду на це, якщо пошук буде виконуватися часто, то більш розумно розглядати час як *час найгіршого випадку*: це буде гарантувати, що на довільних вхідних даних аналогічного розміру код завершить свою роботу за $T(n)$ одиниць часу.

Під *часовою складністю* програми (або алгоритму) $T(n)$ розуміють найбільший час роботи програми (або алгоритму) над вхідними даними довжини n , тобто час найгіршого випадку. Може так бути, що $T(n) = 2^n$, але для деяких даних довжини n треба, наприклад, виконати всього 5 операцій.

Дещо інший підхід полягає в тому, щоб оцінювати час у середньому. Нехай $D_1, \dots, D_k$ — усі дані розміру n , тоді для всіх i від 1 до k знайдемо час $T(D_i)$ роботи над вхідними даними D_i і далі знайдемо середнє арифметичне отриманих значень. Цей й буде *час у середньому*.

$$T_{avg}(n) = \frac{\sum_{i=1}^k T(D_i)}{k}$$

⁶⁵ Припустимо, що ми не збираємось при цьому використовувати бібліотечні функції, а хочемо реалізувати пошук власноруч.

⁶⁶ Наведений синтаксис насправді відповідає типу **list** мови Python.

Оцінка в середньому має сенс. Нехай всі вхідні дані рівномірні. З одного боку, якщо на переважній більшості даних програма (або алгоритм) працює швидко, то незначною кількістю «поганих» випадків можна знехтувати. З іншого боку, якщо програма неодноразово запускається з різними даними, то оцінка часу в середньому буде більш адекватною.

Надалі під $T(n)$ будемо розуміти виключно час найгіршого випадку.

20.5 Асимптотична складність

Час очікування завершення програми більш за все залежить від порядку зростання функції $T(n)$. Наприклад, функція n^3 зростає швидше за n^2 , а функції $3n^2$ і $n^2 + 100n$ зростають з приблизно однаковою швидкістю (конкретні константи і доданки менших порядків суттєво впливають тільки для малих значень n).

Використовується O , Ω і Θ -нотація для позначення складності.

Якщо виконано

$$\exists K, c_2 > 0 \forall n \geq K \ g(n) \leq c_2 f(n)$$

то пишуть $g(n) = O(f(n))$ і кажуть, що функція f є асимптотичною верхньою оцінкою функції g .

Якщо виконано

$$\exists K, c_1 > 0 \forall n \geq K \ c_1 f(n) \leq g(n)$$

то пишуть $g(n) = \Omega(f(n))$ і кажуть, що функція f є асимптотичною нижньою оцінкою функції g .

Якщо виконано

$$\exists K, c_1 > 0, c_2 > 0 \forall n \geq K \ c_1 f(n) \leq g(n) \leq c_2 f(n),$$

то пишуть $g(n) = \Theta(f(n))$. У цьому випадку функція f є асимптотично точною оцінкою функції g .

Якщо $T(n) = O(f(n))$, то функція f є асимптотичною верхньою оцінкою часової складності. Це гарантує, що виконання триватиме не більше ніж $f(n)$ часу за порядком. Можливо, що цю оцінку часу можна покращити (за порядком), якщо, наприклад, більш ретельно провести обрахунки.

Якщо $T(n) = \Omega(f(n))$, то функція f є асимптотичною нижньою оцінкою часової складності. Це означає, що для довільного n знайдуться дані, на яких виконання триватиме не менш ніж $f(n)$ часу за порядком. У цьому випадку оцінку часу $f(n)$ покращити вже не можна, але для деяких випадків часу $f(n)$ може не вистачити.

Якщо $T(n) = \Theta(f(n))$, то функція f є асимптотично точною оцінкою часової складності. Точна оцінка є досяжною (є вхідні дані, на яких вона досягається) і при цьому покращити її вже не можна.

У цій нотації складність реалізованого алгоритму пошуку елемента в масиві складає $\Theta(n)$, оскільки n одиниць часу за порядком буде достатньо, щоб пройти та перевірити весь масив; з іншого боку, завжди можна побудувати такий приклад, коли масив прийдеться проходити до останнього елемента, тобто менш ніж n одиниць часу не вистачить.

Розглянемо ще один приклад. Знайти заданий елемент у відсортованій (за зростанням) послідовності цілих довжини n . Використаємо *алгоритм двійкового пошуку*. Порівнюємо заданий елемент із середнім елементом, якщо вони однакові, то знайшли; якщо середній більше, то продовжимо пошук у лівій частині послідовності; інакше продовжимо пошук у правій частині послідовності.

Рекурсивна версія цього алгоритму виглядає так. Допоміжна рекурсивна функція шукає елемент **sample** у масиві **a** в діапазоні індексів [**i_from**, **i_to**).

```
def _binsearch(a, i_from, i_to, sample):
    if i_from >= i_to:
        return -1
    avg = (i_from + i_to) // 2
    if sample == a[avg]:
        return avg
    elif sample < a[avg]:
        return _binsearch(a, i_from, avg, sample)
    else:
        return _binsearch(a, avg + 1, i_to, sample)
```

Результуюча функція використовує попередню.

```
def binsearch(a, sample):
    return _binsearch(a, 0, len(a), sample)
```

Часова складність роботи рекурсивної функції **_binsearch** над послідовністю довжиною n складає $\Theta(\log n)$.

► На кожному рекурсивному виклику діапазон пошуку зменшується якнайменш удвічі, тому загальна кількість викликів буде $O(\log_2 n)$, але основа логарифму насправді несуттєва, оскільки перехід від однієї основи до іншої виконується множенням на константу, тобто порядок росту не змінюється. Отже, $T(n) = O(\log n)$. З іншого боку, за послідовності $0, 1, \dots, n$ при пошуку елемента 0 нам прийдеться зробити приблизно $\log n$ рекурсивних викликів, тому час $\log n$ покращити не можна. ◀

Часова складність результуючої функції **binsearch** теж складає $\Theta(\log n)$.

Наведений алгоритм можна записати й без рекурсії, замінивши рекурсію ітерацією.

```
def binsearch(a, sample):
    i_from = 0
    i_to = len(a)
    pos = -1
    while i_from < i_to:
        avg = (i_from + i_to) // 2
        if sample == a[avg]:
            i_from = i_to
            pos = avg
        elif sample < a[avg]:
            i_to = avg
        else:
            i_from = avg + 1
    return pos
```

Часова складність ітеративної версії теж складає $\Theta(\log n)$.

Зауваження. Можна оцінювати складність як алгоритму, так й коду, що його реалізує. Але варто пам'ятати, що складність коду за невдалої реалізації може бути більша за складність алгоритму.

20.6 Просторова складність

Аналогічно до витрат часу аналізують витрати пам'яті при роботі програми або алгоритму. Під **просторовою**, або **емнісною, складністю** (англ. *space complexity*) програми (або алгоритму) $S(n)$ розуміють найбільшу необхідну кількість **допоміжної** пам'яті при роботі над вхідними даними розміру n . Аналогічно часової складності в більшості випадків достатньо аналізувати не точні значення функції $S(n)$, а достатньо обмежитися порядком її зростання. Так само розглядають O , Ω і Θ нотацію.

Оцінимо просторову складність написаних функцій у припущенні, що непотрібні об'єкти відразу утилізуються. Насправді це доволі серйозне припущення, що може не мати майже нічого спільного з реальністю.

Функція **search** використовує константну кількість локальних змінних, отже вона вимагає $\Theta(1)$ додаткової пам'яті. Рекурсивна функція **binsearch** може занурюватися в рекурсію на глибину порядку $\log n$ (але й не глибше). Кожен рекурсивний виклик вимагає певної константної кількості пам'яті. Тому для неї необхідно $\Theta(\log n)$ пам'яті. Ітеративна функція **binsearch** використовує константну кількість додаткової пам'яті (тільки 4 додаткові змінні і ця кількість не залежить від розміру вхідних даних), тому для неї просторова складність $\Theta(1)$.

Виходячи з реалій мови Python, слід згадати, що невикористовувані об'єкти (результати проміжних обчислень) можуть не відразу вивантажуватись з пам'яті через оптимізацію роботи збиральника сміття. Але враховувати їх в обсязі пам'яті не варто, адже за нестачі пам'яті вони можуть бути утилізовані збиральником сміття.

20.7 Складність задачі

Для задачі пошуку елемента у відсортованому списку було наведено два алгоритми з різною часовою складністю. Розрізняють складність задачі й складність алгоритму (коду).

Якщо існує алгоритм розв'язання задачі, що працює з часом $O(f(n))$, то кажуть, що задачу можна розв'язати за час $O(f(n))$. Якщо кожен алгоритм розв'язання задачі працює з часом $\Omega(f(n))$, то $\Omega(f(n))$ також є й оцінкою складності цієї задачі. Часова складність задачі становить $\Theta(f(n))$, якщо існує алгоритм, що розв'язує її за час $O(f(n))$, і кожен алгоритм, що її розв'язує, вимагає якнайменш $\Omega(f(n))$ часу; тобто оцінку часу $f(n)$ покращити (за порядком) не можна.

Контрольні запитання

1. Дати означення/стисле тлумачення/короткий опис понять: часова складність програми/алгоритму, просторова складність програми/алгоритму, часова складність задачі, просторова складність задачі, час найгіршого випадку, час у середньому, асимптотична складність, $O(\cdot)$, $\Theta(\cdot)$.

2. У чому полягає емпіричний підхід до оцінки часової складності програми?

3. У чому полягає математичний підхід до оцінки часової складності програми?

Задачі та вправи

1. Розглянемо функцію, що обчислює n -й елемент послідовності.

```
def f(n):  
    if n <= 0:  
        return 0  
    if n == 1:  
        return 1  
    return f(n - 1) + f(n - 2)
```

а) Яку загальновідому послідовність обчислює ця функція?

б) Яка часова складність наведеної функції відносно значення n ?

в) А якщо рахувати складність відносно розміру вхідних даних?

г) Яка просторова складність наведеної функції відносно значення n ?

д) А якщо врахувати специфіку об'єктної моделі мови Python?

е) Яка просторова складність наведеної функції відносно розміру вхідних даних?

ж) Чи можна цю задачу розв'язати швидше та з меншими витратами пам'яті? Якщо можна, то як? Якими тоді будуть часова та просторова складності?

2. Яка часова складність обчислення **s.insert(i, el)** для списку **s**?
3. Яка часова складність обчислення **s.append(el)** для списку **s**?
4. Яка часова складність виконання інструкції **del s[i]** для списку **s**?
5. Яка часова складність виконання інструкції **del s[-1]** для списку **s**?
6. Розглянемо функцію, що виконує перетворення списку

```
def f(s, el):  
    while s and s[0] != el:  
        del s[0]
```

Яка часова складність наведеної функції?

7. Розглянемо функцію, що виконує перетворення списку

```
def f(s, el):  
    i = 0  
    while i < len(s):  
        if s[i] == el:  
            del s[i]  
        else:  
            i += 1
```

Яка часова складність наведеної функції?

8. Розглянемо фрагмент коду, в якому змінна **s** позначає список рядків.

```
res = ''  
sep = ';' '  
for el in s:  
    res += sep + el
```

Яка часова складність наведеної функції?

Предметный показчик

**kwargs.....	277
*args.....	277
:=.....	161
__init__.py.....	145
add.....	255
all.....	244
and.....	94
any.....	244
ArithmeticError.....	117
BaseException.....	116, 117
break.....	169
builtins.....	131
case.....	105
clear.....	203
comprehension.....	234
continue.....	169
copy.....	197
count.....	196
decode.....	214
DEDENT.....	47
deepcopy.....	197, 265
del.....	56, 203, 263
dict.....	258
dict.....	252
difference.....	254
difference_update.....	254
discard.....	255
elif.....	99
else.....	99, 162, 163
encode.....	214
endswith.....	211
enumerate.....	205
except.....	116
Exception.....	117
extend.....	203
False.....	31
filter.....	243
find.....	211
for.....	163
from.....	142, 144
fromkeys.....	267
frozenset.....	257
frozenset.....	252
get.....	261
global.....	135
globals.....	268
hash.....	250
id.....	93
if.....	99
import.....	139, 140, 142
INDENT.....	47
index.....	200
IndexError.....	194
input.....	71
insert.....	203
intersection.....	254
intersection_update.....	254
is.....	92
is not.....	93
isalnum.....	210
isalpha.....	209
isascii.....	209
isdecimal.....	209
isdigit.....	209
isdisjoint.....	255
isinstance.....	93
islower.....	208
isspace.....	210
issubset.....	254
issuperset.....	254
isupper.....	208
items.....	260
iter.....	229, 231
join.....	213
KeyboardInterrupt.....	117
KeyError.....	255
keys.....	260
len.....	193, 265
list.....	191
locals.....	268
lower.....	208
lstrip.....	210
map.....	243
match.....	105
max.....	242
min.....	242
NameError.....	117
NEWLINE.....	46
next.....	229, 230, 232
nonlocal.....	135
not.....	94
Нип даних.....	29
or.....	94
partition.....	213
pickle.....	232
pop.....	203, 264
popitem.....	264
print.....	69
raise.....	120, 122
range.....	163, 164
remove.....	255
removeprefix.....	211
removesuffix.....	211
reverse.....	204
reversed.....	205
rpartition.....	213
rsplit.....	212
rstrip.....	210
scope.....	129
set.....	252
setdefault.....	263
sort.....	204
sorted.....	205
split.....	212
splitlines.....	213
startswith.....	211
stderr.....	226
stdin.....	226
stdout.....	226
StopIteration.....	229
strip.....	211
suite.....	74
sum.....	242
symmetric_difference.....	254
symmetric_difference_update.....	254
sys.modules.....	139, 268
sys.path.....	140
True.....	31

try	116	Системи запису	57
tuple	191	Індекс	190
type	33	Інструкція	11
TypeError	117	break	169
Unicode	25	continue	169
union	254	del	56, 263
update	254, 264	for	163
upper	208	global	135
US-ASCII	25	if	99
UTF-8	26	import	139, 141, 142, 144
ValueError	116, 117	match	104
values	260	nonlocal	135
while	158, 162	pass	76
yield	237	raise	120, 122
yield from	239	return	74, 78
ZeroDivisionError	117	try	116, 119
zip	244	else	120
Абстракція	67	except-секція	120
Алгоритм	11	finally	120
алгоритм двійкового пошуку	283	try-секція	119
Алфавіт	40	while	158, 162
Байт-код	13	yield	237
Бібліотека	67	Виразу	70
math	73	Означення функції	75
Системна	69	Присвоювання	53
Стандартна	69	Проста	74
Стороння	69	Розгалуження	99
Блок	75	Складена	74
Блок, коду	130	Умовна	99
Вбудовані засоби	69	Циклу	
Виняткова ситуація	114	З передумовою	158
Виняток	114	Інтерпретатор	12
Генерування	114	Ітератор	228
Оброблення	114	Ітерування	164
Перехоплений	117	ключ	251
Вираз	34	Ключові слова	41
Умовний	96	Код	
Відрізки послідовностей	199	Дійсного числа	
Глибина рекурсії	154	Денормалізовані числа	24
Дані		З фіксованою крапкою	23
Вихідні	11	З плаваючою крапкою	23
Вхідні	11	Нечисла	24
Проміжні	11	Цілого числа	
Декомпозиція	67	Беззнаковий	21
Доступ		Додатковий	22
Послідовний	191	Обернений	22
Прямий	190	Прямий	22
Дублювання коду	67	Код, що охороняється	116
Завантажувач	13	Команда	11
Зв'язування		Командний рядок	225
Лівобічне	35	Коментар	45
Правобічне	35	Надлишковий	46
змінна		Комп'ютерна програма	9
операції	52	Компілятор	12
Змінна		Констант	42
Вільна	132	Контейнер	190
Глобальна	79, 131	Асоціативний	251
Ім'я	52	Послідовнісний	190
Локальна	79, 131	Копіювання	
Тип	52	Глибоке	197
у Python	55	Неглибоке	196
Ідентифікатор	41	Кросплатформеність	14
Інтегроване	13	Ледачі обчислення	94
Імпортуння	139	Лексема	40
Ім'я		Літерал	42
Змістовне	57	Лямбда-функція	240
Кваліфіковане	73	Масив	190
Розв'язання	Див. Розв'язання імен	Мова	10

Машинна	12	Дно рекурсії	152
Природна	10	Пряма	152
Програмування	10, 12	Розв'язання імен	134
Високого рівня	12	Розмотування стеку	123
Динамічна типізація	52	Рядок	43
Низького рівня	12	f-рядки	45
Статична типізація	52	Довжина	43
Типізована	52	Рядок вхідного тексту	
Семантика	10	Логічний	46
Синтаксис	10	Фізичний	46
Штучна	10	Рядок документації	83
Модель	10	Система числення	18
Модуль	67	Позиційна	18
Набір інструкцій	См. suite	Складність	
Наповнювання	234, 253, 260	Просторова	284
Об'єкт	50	Часова	282
callable	231	Сміття	55
hashable	250	Збирання	55
iterable	230	Структура даних	190
Обернена сумісність	15	Тестування	108
Об'єкт		Валідація	108
Змінюваний	51	Верифікація	108
Значення	50	Покриття коду	108
Зображуване значення	51	Тип даних	
Контейнер	50	bool	29
Незмінюваний	51	complex	29
Сталий	51	float	29
Область видимості	129	int29	
Область дії означення	129	Перетворення типу	37
Оператор	34, 41	Скалярний	29
Адитивний	35	Складений	29
Бінарний	34	Сумісність	36
Логічний	94	Транслятор	12
Мультиплікативний	35	Умова	96
Порівняння	92	продовження	159
Присвоювання	161	Умовна операція	101
Пріоритет	35	Формат дійсного числа	
Унарний	34	З фіксованою крапкою	22
Операція	34	Нормалізований експоненційний	22
булева	94	Фрейм виконання	131
Операція зрізу	201	Функціональне проєктування	88
Пакет	67, 145	Функція	68
Парадигма програмування	9	Анонімна	240
Декларативна	9	Аргументи виклику	68, 74
Імперативна	9	Іменовані	71
Структурне програмування	75	Ключові	71
Параметри програми	225	Позиційні	70
Побічний ефект	135	Виклик	68
Правило 80/20	115	Довга	86
Принцип абстракції даних	30	Заголовок	74
Принцип єдиної відповідальності	Див. функціональне проєктування	Тіло	74
Принцип найменшого здивування	185	Фактичні параметри	74
Присвоювання	53	Формальні параметри	74
Складене	58	Значення за умовчанням	80
Цільовому списку	58	Хеш-функція	249
Програмування	9	Цикл	
Простір імен	129	for	163
Глобальний	131	while	158
Локальний	131	Заголовок	158
Процес	11	Інваріант	161
Рекурентне співвідношення	166	Ітерація	158
Початкові умови	166	Лічильник	165
Рекурсивна функція	152	Тіло	158, 162
Рекурсивне означення	151	Умова продовження	159
Рекурсія		Час найгіршого випадку	282
Взаємна	152	Час у середньому	282
Глибина	154	Шаблон проєктування	228

Список використаних джерел

1. Python documentation. URL: <https://docs.python.org>
2. Welcome to Python.org. URL: <https://www.python.org>
3. Ставровський А.Б., Карнаух Т.О. Програмування. Перші кроки. М.: Видавничий дім “Вільямс”, 2005. 400 с.
4. Белов Ю.А., Карнаух Т.О., Коваль Ю.В., Ставровський А.Б. Вступ до програмування мовою С++. Організація обчислень. К.: ВПЦ "Київський університет", 2012. 175 с.
5. Карнаух Т.О., Коваль Ю.В., Потієнко М.В., Ставровський А.Б. Вступ до програмування мовою С++. Організація даних. К.: ВПЦ "Київський університет", 2015. 151 с.
6. Веклич Р.А., Карнаух Т. О., Ставровський А. Б. Вступ до програмування мовою С++. Структури даних. К. : ВПЦ "Київський університет", 2018. 99 с.
7. IEEE Standard for Floating-Point Arithmetic. [Revision of IEEE Std 754—2019]. ISBN 978-1-5044-5924-2 2019, DOI:10.1109/IEEESTD.2019.8766229.
8. Fowler M. Refactoring: Improving the Design of Existing Code. Boston, MA: Addison-Wesley, 2000. 337 p.
9. Fild A.J., Harrison P.G. Functional programming. Addison-Wesley, 1988. 616 p.