

**Національний університет «Чернігівський колегіум»
імені Т.Г. Шевченка**

А.О. Костюченко

**ОСНОВИ ПРОГРАМУВАННЯ
МОВОЮ PYTHON**

Навчальний посібник

Чернігів, 2020

УДК 004.438 Python
ББК 32.973.26-018.1

ORCID 0000-0002-6178-6444

K72 Костюченко А.О.

Основи програмування мовою Python: навчальний посібник. Ч.: ФОП Баликіна С.М., 2020. 180 с.

У посібнику розкриті фундаментальні принципи програмування мовою Python. Описано базовий синтаксис мови Python 3: типи даних, оператори, умови, цикли, вбудовані функції, обробка виключних ситуацій, часто використовувані модулі стандартної бібліотеки. Для кращого засвоєння матеріалу в посібнику наведена велика кількість прикладів програмного коду, що забезпечить можливість навчатися програмувати мовою Python самостійно. Весь матеріал ретельно підібраний, добре структурований та компактно викладений, що дозволяє використовувати посібник як зручний довідник.

Посібник складений з урахуванням досвіду викладання програмування і розрахований на учнів, студентів та інших читачів, що починають вивчати програмування мовою Python. Посібник також буде корисний вчителям ЗСО та викладачам ЗВО.

Рецензенти:

Покришень Дмитро Анатолійович - кандидат педагогічних наук, доцент, завідувач кафедри природничо-математичних дисциплін та інформаційно-комунікаційних технологій в освіті Чернігівського обласного інституту післядипломної педагогічної освіти імені К.Д. Ушинського

Цибко Ганна Юхимівна - кандидат педагогічних наук, доцент кафедри інформатики і обчислювальної техніки Національного університету «Чернігівський колегіум» імені Т.Г.Шевченка

Рекомендовано до друку вченою радою Національного університету «Чернігівський колегіум» імені Т.Г.Шевченка, протокол № 6 від 05 лютого 2020 р.,

© Костюченко А.О., 2020

ЗМІСТ

ВСТУП.....	6
1. Знайомство з мовою програмування Python	7
1.1. Історія мови	7
1.2. Особливості мови	8
1.3. Застосування мови	13
2. Основи мови Python.....	15
2.1. Синтаксис мови.....	15
2.2. Запуск програм.....	16
2.3. Змінні та літерали	21
2.4. Типи даних	22
2.4.1 Вбудовані типи даних.....	23
2.4.2. Ініціалізація змінних.....	24
2.4.3. Змінювані і незмінювані типи	26
2.5. Введення виведення даних.....	28
3. Числові дані.....	30
3.1. Робота з цілими та дійсними числами	31
3.1.1. Математичні та бітові операції.....	31
3.1.2. Порядок обчислення операцій.....	33
3.1.3. Вбудовані функції цілих і дійсних чисел	34
3.1.4. Модуль math.....	37
3.2. Приклади розв'язування задач	43
3.3. Робота з комплексними числами	44
4. Винятки та їх обробка	45
5. Організація розгалужень в програмах	52
5.1. Логічні вирази і логічний тип даних	52
5.2. Оператори відношень (порівнянь)	52
5.3. Умовний оператор if-else (if-elif-else)	53
5.4. Тримісний оператор if/else	57
5.5. Логічні оператори	57
5.6. Приклади розв'язування задач	60
6. Циклічні оператори	61
6.1. Цикл з передумовою (Цикл while)	62
6.1.1. Приклади розв'язування задач	63
6.2. Тип діапазон (range).....	64
6.3. Цикл for.....	64
6.3.1. Приклади розв'язування задач	65

6.4. Інструкції управління циклами.....	66
6.5. Блок else в циклах	68
6.6. Вкладені цикли.....	69
7. Структури даних	70
7.1. Списки	71
7.1.1. Задання списків	71
7.1.2. Доступ до елементів списку. Зрізи.....	74
7.1.3. Зміна та вилучення елементів списку	77
7.1.4. Змінюваність типу список.....	79
7.1.5. Методи списків	81
7.1.6. Порівняння списків.....	83
7.1.7. Вкладені списки	84
7.1.8. Приклади розв'язування задач	86
7.2. Кортежі	90
7.2.1. Задання кортежів	90
7.2.2. Виконання дій над кортежами та їхніми елементами	92
7.3. Словники	94
7.3.1. Створення словників	95
7.3.2. Виконання дій над елементами словника	95
7.3.3. Методи словників	97
7.3.4. Приклади розв'язування задач	100
7.4. Рядкові величини	100
7.4.1. Рядкові літерали та їх задання	101
7.4.2. Задання рядків	102
7.4.3. Доступ до символів рядку. Зрізи	105
7.4.4. Виконання дій над рядками та їхніми елементами	108
7.4.5. escape-послідовності.....	109
7.4.6. Методи рядків	110
7.4.7. Приклади розв'язування задач	119
7.5. Множини	121
7.5.1. Задання множини	121
7.5.2. Виконання дій над елементами множини.....	122
7.5.3. Порівняння множин.....	123
7.5.4. Методи множин	124
7.5.5. Приклади розв'язування задач	126
8. Функції.....	127
8.1. Опис та виклик функцій.....	128
8.2. Розширене використання параметрів та аргументів.....	131

8.2.1. Значення параметрів за замовчуванням.....	131
8.2.2. Ключові аргументи.....	133
8.2.3. Змінна кількість аргументів.....	134
8.2.4. Обов'язкові ключові аргументи.....	137
8.3. Глобальні та локальні змінні.....	138
8.3.1. Глобальні змінні.....	138
8.3.2. Локальні змінні.....	138
8.3.3. Зв'язок однойменних локальних і глобальних змінних.....	139
8.3.4. Нелокальні змінні.....	140
8.4. Правила локалізації.....	141
8.5. Lambda функції.....	142
8.5. Рекурсія.....	143
8.6. Приклади розв'язування задач.....	144
9. Файли.....	146
9.1. Відкриття та закриття файлу.....	147
9.2. Атрибути файлового об'єкта.....	149
9.3. Читання з файлу.....	149
9.4. Запис у файл.....	151
9.5. Додаткові методи роботи з файлами.....	152
9.6. Використання менеджера контексту.....	153
9.7. Приклади розв'язування задач.....	154
Список використаних джерел.....	157
Додаток 1. Імпортування модулів та їх атрибутів.....	158
Додаток 2. Модуль random.....	161
Додаток 3. Форматування рядків.....	165
Додаток 4. Дерево класів Exception.....	174

ВСТУП

В основу пропонованого навчального посібника покладено цикл занять, проведених автором для студентів природничо-математичного факультету Національного університету «Чернігівський колегіум» імені Т.Г. Шевченка.

Навчання програмування займає досить вагоме місце в освітньому процесі. Досвід багатьох країн показує, що вивчення правил кодування і мов програмування сприяє розвитку логічного та креативного мислення. Також варто відмітити те, що програмування навчає творчості, шляхом набуття навичок шукати рішення, оскільки для розв'язання практичних задач можна підійти з різних сторін.

Взагалі, мов програмування досить багато. Більш того, час від часу з'являються нові мови. Тому природним чином виникає питання: чому саме Python? Тут можна видалити кілька пунктів.

Мова програмування Python - це мова високого рівня, досить "молода", але дуже популярна, яка вже зараз широко використовується на практиці і сфера застосування Python постійно розширюється.

Мова Python бурхливо розвивається. Цьому сприяє не тільки досить вдала концепція мови, але також сформоване згуртоване співтовариство розробників і популяризаторів мови.

Синтаксис мови Python мінімалістичний і гнучкий. На цій мові можна складати прості та ефективні програми.

Стандартна бібліотека для цієї мови містить безліч корисних функцій, що значно полегшує процес створення програмних кодів.

Мова Python підтримує кілька парадигм програмування, включаючи структурну, об'єктно-орієнтовану і функціональну.

Мова Python цілком вдалий вибір для першої мови при навчанні програмуванню

Важливий і той факт, що необхідне програмне забезпечення, включаючи середовища розробки, в основній своїй масі безкоштовні.

Все це дає підстави розглядати Python в якості одного з найбільш перспективних мов програмування.

1. Знайомство з мовою програмування Python

1.1. Історія мови

Автором мови Python є нідерландський програміст Гвідо ван Россум. На той час він був співробітником голландського інституту CWI і працював над мовою ABC[en], яка мала стати максимально простою і зрозумілою декларативною мовою. Мова ABC була достатньо цікавою, але вона мала значний недолік: нею ніхто не користувався. Тому в грудні 1989 року Гвідо розпочав свій «хобі проект» Python, в основу якого лягли окремі елементи мови ABC (фактично, Python створювався як спроба виправити помилки, допущені при проектуванні ABC). Датою появи мови Python вважається 20 лютого 1991 року.

До недавнього часу Гвідо ван Россум був не лише автором мови, а і «доброчинним пожиттєвим диктатором» («Benevolent Dictator For Life») тобто мав право приймати остаточні рішення щодо мови. Проте 12 липня 2018 року він вирішив залишити цей «пост».

Як зазначає сам Гвідо ван Россум, мова названа на честь британського телешоу «Літаючий цирк Монти Пайтона» (англ. *Monty Python's Flying Circus*), хоча інколи мову називають «Пітон».

З'явившись порівняно пізно, Python створювався під впливом багатьох мов програмування [1, 4, 9]:

- ABC – на відміну від значної кількості мов програмування, в мові ABC відразу використовувалися відступи (поля) замість операторних дужок для групування операторів. Такий же принцип був обраний і для Python;
- Modula-3 – популярна в академічних кругах, але майже без практичного програмування мова була однією з перших, де були винятки (try ... except), які і були взяті для Python;
- C, C++ – для Python були взяті деякі синтаксичні конструкції (як пише сам Гвідо ван Россум - він використовував найбільш несуперечливі конструкції з C, щоб не викликати неприязнь у C-програмістів до Python);
- Smalltalk – елементи об'єктно-орієнтованого програмування;

- Lisp – окремі риси функціонального програмування;
- Fortran – зрізи масивів, комплексна арифметика;
- Miranda – спискові вирази;
- Java – модулі logging, unittest, threading (частина можливостей оригінального модуля не реалізована), xml.sax стандартної бібліотеки, спільне використання finally та except при обробці винятків, використання @ для декораторів;
- Icon – генератори.

На даний момент існують дві актуальні версії мови: Python 2.x (остання версія на момент написання – 2.7) і Python 3.x (остання версія на момент написання – 3.8). Це пов'язане з тим, що за свій час «життя» мова Python 2.x акумулювала певну кількість недоліків, які не можна було виправити, не зламавши чийсь код. Основний недолік пов'язаний з рядками (спочатку рядки були байтовими, потім було додано підтримку Юнікоду (Unicode) з неявним приведенням).

Тому в більшості випадків Python 2.x використовується для забезпечення сумісності з раніше написаними програмами. Проте підтримка гілки 2.x буде завершена в січні 2020 року.

Python 3.0 обернено не сумісний з попередньою серією 2.x. Код Python 2.x швидше за все буде видавати помилки при виконанні в Python 3.x. Динамічна типізація Python, разом з планами зміни декількох методів словників, робить механічне портування з Python 2.x в Python 3.0 дуже складним. [1, 2, 6]

Тому нові проекти варто створювати на Python 3.x, якщо немає явних причин використовувати Python 2.x.

1.2. Особливості мови

Мову Python можна охарактеризувати як: інтерпретовану високорівневу мову програмування загального призначення. В різній літературі можна зустріти набагато більше складових мови, які вказують при її описі, тому просто сформулюємо її особливості.

Проста і мінімалістична мова

Читання хорошої програми на Python дуже нагадує читання англійського тексту, хоча і досить строгого. Така псевдокодова природа Python є однією з його найсильніших сторін. Python надає можливість програмісту зосередитися на розв'язуванні задачі, а не на самій мові. Ядро мови без додаткових модулів досить просте і мінімальне - це надає мові простоти та логічності, що відповідно спрощує її вивчення.

Мова легка в освоєнні та простота для вивчення

Мовою Python надзвичайно легко почати програмувати. Python має виключно простий синтаксис. Досить часто в закладах освіти обирають Python як першу мову програмування для своїх студентів.

Вільна і відкрита мова

Python - це приклад вільного і відкритого програмного забезпечення – FLOSS (Free / Libre and Open Source Software). Простіше кажучи, ви маєте право вільно використовувати копії цього програмного забезпечення, переглядати його вихідні тексти, вносити зміни, а також використовувати його частини в своїх програмах. Python був створений і постійно поліпшується співтовариством, яке просто хоче зробити його краще.

Високорівнева мова

Машина (комп'ютер, процесор) розуміє лише певний обмежений та примітивний набір спеціальних команд. Коли пишеться код деякою мовою, по суті, вказуються інструкції, які повинна виконати машина. В залежності від того, які ці інструкції, мови і поділяються на високорівневі та низькорівневі. Якщо код пишеться в машинних інструкціях (командах), то це низькорівневі мови. Якщо ж додатково необхідне певне програмне забезпечення, яке буде переводити код, написаний мовою програмування, в машинні інструкції (команди) то це високорівнева мова програмування. Програмне забезпечення, за яким відбувається вказане переведення, називається компіляторами або інтерпретаторами.

Мова загального призначення

Мова призначена для розв'язування досить широкого спектру задач.

Кросплатформенна мова

Завдяки своїй відкритій природі, Python був портований на багато платформ (змінений таким чином, щоб працювати на різних пристроях та різних операційних системах). Один і той самий код буде запускатися на різних платформах без будь-яких змін за умови, що в коді відсутні системно-залежні функції.

Python можна використовувати в GNU / Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE і навіть на PocketPC.[1, 8, 10]

Інтерпретована мова

Як зазначалося раніше, для виконання програми на комп'ютері необхідно команди мови програмування перетворити у команди, які здатен виконувати процесор комп'ютера. Існує два типи таких перетворень – інтерпретація і компіляція, які здійснюються спеціальними програмами-трансляторами.

Програма, написана компільованою мовою програмування, як наприклад, Pascal або C++, аналізується і при відсутності помилок перетворюється, за допомогою компілятора, в мову, зрозумілу комп'ютеру (бінарний код), після чого записується в оперативну пам'ять комп'ютера або на диск. Утворений на диску файл є виконуваним файлом і може бути виконаний засобами операційної системи без повторного використання транслятора.

При інтерпретації транслятор аналізує кожен команду програми окремо і, якщо команда не містить синтаксичних помилок, перетворює її в машинний код і відразу виконує. [7] Проте в Python цей процес дещо складніший. Після запуску програми Python аналізує кожен команду програми і, якщо команда не містить синтаксичних помилок, перетворює її в байт-код (машино-незалежний код низького рівня). Далі байт-код передається віртуальній машині Python, яка і виконує інструкції байт-коду, перетворюючи їх в машинний код.

Коли байт-код сформовано, його можна виконати на різних інтерпретаторах, що підтримують цей байт-код. Так, для Python можна виділити такі інтерпретатори:

- CPython – інтерпретатор, реалізований на С. Вважається еталонною реалізацією Python і підтримує більшість активно використовуваних платформ;
- IronPython – Python для .NET Framework і Mono. Компілює Python програми в MSIL (Microsoft Intermediate Language), таким чином надаючи повну інтеграцію з .NET-системою;
- Jython – реалізація Python, що використовує JVM (Java Virtual Machine);
- Stackless – патчі до CPython, який надає розширені можливості багатопотокового програмування і значно більшу глибину рекурсії;
- PyPy – реалізація Python, написана на RPython. В PyPy вбудований трасуючий JIT-компілятор (Just-in-time compilation), який може перетворювати код Python в машинний код під час виконання програми.

Мультипарадигменна мова

В основі Python лежить декілька парадигм програмування: об'єктно-орієнтовна, імперативна, функціональна (функційна), структурна. Проте досить часто про Python говорять зокрема як про об'єктно-орієнтовну мову програмування. В об'єктно-орієнтованих мовах програмування програми будуються на основі об'єктів, які об'єднують в собі дані і функціонал. Об'єктів в мові Python багато, можна навіть сказати, що все є об'єкт (це не дуже коректно, але допустимо).

Розширювана мова

Якщо потрібно, щоб деяка критична частина програми працювала дуже швидко, або є необхідність приховати частину алгоритму, можна написати цю частину програми мовою С або С++, а потім викликати її з програми, написаною мовою Python.

Вбудовувана мова

Код Python можна легко вбудувати в програми на Java, C/C++, щоб надати користувачам можливість написання власних сценаріїв. Взаємодія Python-додатків з іншими системами можлива також за допомогою CORBA (Common Object Request Broker Architecture), XML-RPC (Extensible Markup Language Remote Procedure Call), SOAP (Simple Object Access Protocol), COM (Component Object Model).

Мова з "широкою" стандартною бібліотекою

Стандартна бібліотека Python досить велика і її використання надає можливість програмістам розв'язувати досить різноманітні задачі: з використанням регулярних виразів, перевіркою блоків коду, розпаралелюванням процесів, базами даних, веб-браузерами, CGI, FTP, електронною поштою, XML, XML-RPC, HTML, WAV файлами, криптографією, GUI (графічний інтерфейс користувача) та іншими системно-залежними речами. Крім стандартної бібліотеки, існує безліч інших високоякісних бібліотек, які можна знайти в каталозі пакетів Python

Таким чином, Python – потужна мова, в якій поєднано продуктивність та значні можливості, що робить написання програм одночасно цікавим і легким.

Незважаючи на досить багато переваг, Python має і ряд недоліків, а саме:

- Низька швидкодія в порівнянні з компільованими мовами. Це пов'язане з інтерпретованістю мови, тобто переведенням коду програми в машинний код в процесі роботи програми. Цей недолік частково знімається збереженням модулів у вигляді байт-коду (файли .рус і .pyo).
- В Python відсутня можливість модифікувати вбудовані класи, такі, як int, str, float, list та інші. Однак це дозволяє Python споживати менше оперативної пам'яті і швидше працювати.
- Глобальне блокування інтерпретатора GIL (Global Interpreter Lock) або проблеми з багатопоточністю. В кожен момент часу може виконуватися тільки одна нитка (потік) Python коду, навіть якщо на комп'ютері є кілька процесорів або процесорних ядер. Це зроблено для

унеможливлення руйнування даних при спільній модифікації з різних потоків, оскільки в своїй роботі інтерпретатор Python постійно використовує велику кількість нитко-небезпечних даних. Проте GIL-проблема притаманна CPython, Stackless та PyPy, але відсутня в Jython та IronPython.

1.3. Застосування мови

Python є не лише академічною мовою програмування, на прикладі якої можна тільки навчати основам програмування і далі про неї забути. Серед областей, де активно застосовується мова Python, можна виділити: системне програмування, розробку динамічних веб-сайтів, інтеграцію компонентів, розробку програм для роботи з базами даних, швидке створення прототипів, розробку програм для наукових обчислень, навчання, розробку ігор.

Розробка прикладного програмного забезпечення та ігор.

- BitTorrent – програми для файлообміну. Всі версії до 6-ї були написані мовою Python. Версія 6 була переписана мовою C++.
- Blender – вільний, професійний пакет для створення тривимірної комп'ютерної графіки. Python використовується як засіб створення інструментів і прототипів, системи логіки в іграх, як засіб імпорту / експорту файлів, автоматизації завдань.
- GIMP – растровий графічний редактор. Python використовується для написання додаткових модулів (фільтрів).
- Maya – система тривимірного моделювання та створення мультиплікації підтримує інтерфейс для управління з сценаріїв мовою Python
- Civilization IV. Велика частина гри написана мовою Python.
- World of Tanks. Як скриптова мова в проекті використовується Python.
- LibreOffice. Написання макросів.

Web-розробка.

- Яндекс. Python використовується в ядрі Яндекс диску.
- DropBox. Сервіс розроблений мовою Python.
- Instagram. Розроблений мовами Python і Django.

- Компанія Google широко використовує Python в своїй пошуковій системі.
- Служба колективного використання відеоматеріалів YouTube в значному ступені реалізована мовою Python.
- Реалізація поштового сервера IronProt використовує понад 1 мільйон рядків програмного коду мовою Python.
- Мовою Python написані багато веб-фреймворків: App Engine, Pylons, TurboGears, CherryPy, Flask, Pyramid та інші.

Компанії, які використовують Python.

- iRobot використовує Python в розробці комерційних роботизованих пристроїв.
- ESRI використовує Python в якості інструменту налаштування своїх популярних геоінформаційних програмних продуктів під потреби кінцевого користувача.
- NSA використовує Python для шифрування і аналізу розвідданих.
- EVE Online і Massively Multiplayer Online Game (MMOG) широко використовують Python в своїх розробках.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm і IBM, використовують Python для тестування апаратного забезпечення.
- Industrial Light & Magic, Pixar і інші використовують Python у виробництві анімаційних фільмів.
- JPMorgan Chase, UBS, Getco і Citadel застосовують Python для прогнозування фінансового ринку.
- NASA, Los Alamos, Fermilab, JPL і інші використовують Python для наукових обчислень.

Системне адміністрування. Python встановлюється за замовчуванням на всі Linux-сервери і використовується в автоматизації роботи системного адміністратора.

Вбудовані системи. Дуже часто Python використовується для програмування вбудованих систем. Найвідоміший проект, який використовує Python - це Raspberry Pi.

2. Основи мови Python

2.1. Синтаксис мови

Синтаксис мови Python, як і сама мова, достатньо простий. Можна виділити наступні твердження.

- Кінець рядка є кінцем інструкції (прикінцеві символи непотрібні).
- Для надання значень змінним оператором присвоювання є знак дорівнює «=». Формат оператора: ім'я_змінної=вираз.
- Основна інструкція та вкладені інструкції (вкладений блок інструкцій) записуються відповідно до одного шаблону: основна інструкція завершується двокрапкою, наступними рядками розташовуються вкладені інструкції, з однаковим відступом на початку рядків по відношенню до основної інструкції. Наприклад:

Основна інструкція:

Вкладений блок інструкцій

Тобто вкладені інструкції об'єднуються в блоки за величиною відступів. Відступ може бути будь-яким, головне, щоб в межах одного вкладеного блоку відступ був однаковий. Не варто забувати про читабельність коду, так, відступ в 1 пропуск є малочитабельним. Рекомендується використовувати 4 пропуски.

Можна бачити, що синтаксис оформлення основної інструкції та вкладеного блоку інструкцій істотно відрізняється від синтаксису більшості мов, в яких використовуються операторні дужки для виділення вкладеного блоку інструкції (наприклад, `begin ... end` в Паскалі або `{ ... }` в Сі).

- Розмір літер має значення, тобто великі і маленькі літери вважаються різними. Більшість службових слів (окрім: `False`, `None`, `True`) та вбудованих функцій пишуться маленькими літерами.

Проте існує декілька спеціальних випадків:

- Можна записати кілька інструкцій в одному рядку, розділяючи їх крапкою з комою:

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

- Можна записувати одну інструкцію в декілька рядків. Для цього необхідно розмістити її в парі круглих, квадратних або фігурних дужок:

```
if (a < 1 and b < 2 and
    c < 3 and d < 4):
    print('spam' * 3)
```

- Якщо тіло вкладеної інструкції містить єдиний оператор, то він може розташовуватися в тому ж рядку, що і основна інструкція:

```
if x > y: print(x)
```

Крім конструкцій мови, програма може містити коментарі.

Коментар – це довільний текст у будь-якому місці програми, що пишеться після символу #, і представляє інтерес лише як замітка для того, хто буде переглядати код програми.

Наприклад:

```
print('Привіт, Світ!') # print -- це функція
або:
# print -- це функція
print('Привіт, Світ!')
```

Коментарі містять пояснювальні тексти і полегшують читання і розуміння програм, тому намагайтеся в своїх програмах писати якомога більше корисних коментарів, які містять: припущення, важливі рішення, важливі деталі, проблеми, які ви намагаєтеся вирішити, проблеми, яких ви намагаєтеся уникнути і т.д.

2.2. Запуск програм

Виконання програм на комп'ютері здійснюється операційною системою. В завдання операційної системи входить виділення ресурсів (оперативної пам'яті та ін.) для програми, заборона або дозвіл на доступ до пристроїв введення / виведення і т.д.

Для запуску програм, написаних мовою Python, окрім операційної системи необхідна програма-інтерпретатор (віртуальна машина) Python (Рис. 2.1). Як зазначалося раніше, ця програма приховує від Python-програміста всі особливості операційної системи, тому, написавши програму

мовою Python в системі Windows, її можна запустити, наприклад, в GNU/Linux і отримати такий же результат її виконання.

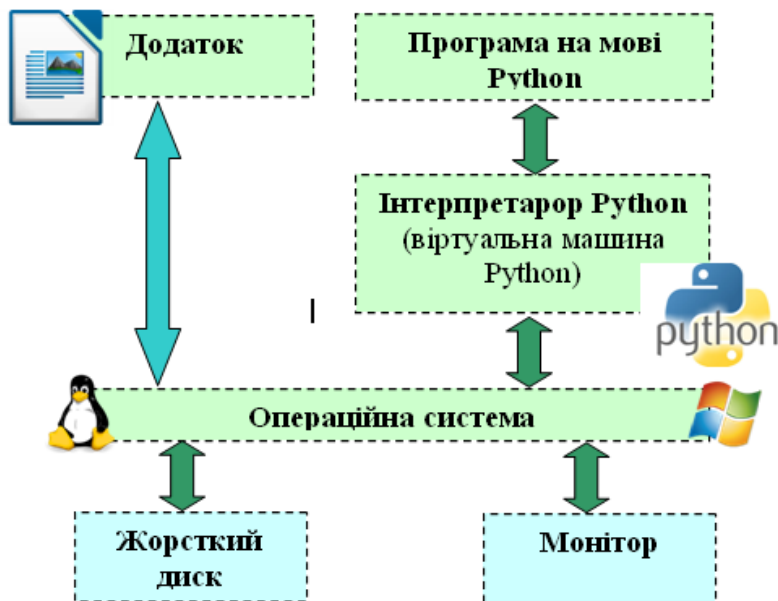


Рис. 2.1. Схема орієнтовної роботи програмі на мові Python

Існує два способи запуску програм, написаних мовою Python: використання інтерактивного режиму інтерпретатора (інтерактивний режим) та використання файлу з текстом програми (пакетний режим)

Інтерактивний режим роботи

Подібно мовам Lisp та Prolog в режимі налагодження, інтерпретатор Python має інтерактивний режим роботи, при якому введений з клавіатури вираз відразу ж виконується, а результат виконання виводиться на екран. Цей режим може бути цікавий не тільки початківцям, але й досвідченим програмістам, які мають можливість протестувати в інтерактивному режимі будь-який фрагмент коду, перш ніж використовувати його в основній програмі.

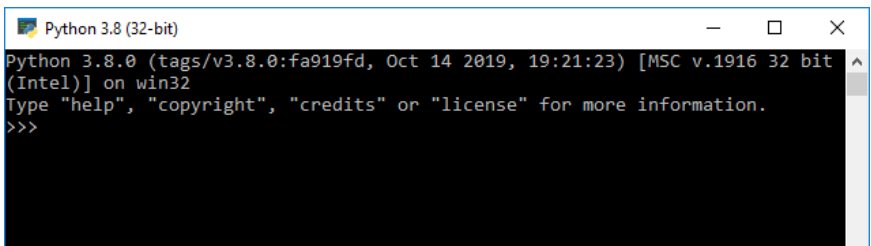
Для запуску інтерпретатора Python в інтерактивному режимі необхідно в терміналі чи командному рядку ввести команду `python` або `python3` та

натиснути Enter. Користувачі Windows можуть запустити інтерпретатор в командному рядку, якщо належним чином встановлено змінну PATH або з директорії (папки), в якій встановлено Python.

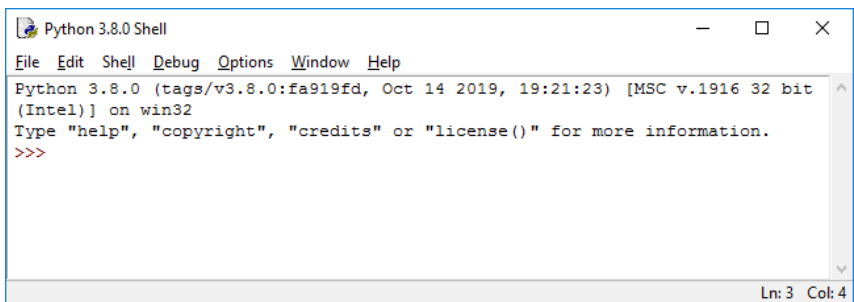
В комплекті разом з інтерпретатором Python йде IDLE (Integrated Development and Learning Environment - інтегроване середовище розробки). За своєю суттю воно подібне до інтерпретатора, запущеного в командному рядку, але має розширений набір можливостей (підсвічування синтаксису, перегляд об'єктів, налагодження і т.п.).

Для запуску IDLE в Windows необхідно вибрати ярлик "IDLE (Python 3.x GUI - XX bit)" з списку встановлених програмних засобів в меню "Пуск". В GNU Linux оболонка IDLE за замовчуванням відсутня, тому її попередньо потрібно встановити. Також можна скористатися online консоллю для Python (<https://www.python.org/shell/>).

В результаті запуску Python в інтерактивному режимі буде виведено запит командного рядка в вигляді символів «>>>». Після вказаних символів і можна набирати команди для виконання (Рис. 2.2).



а) текстове вікно інтерактивного режиму



б) графічне вікно інтерактивного режиму

Рис. 2.2. Вікно інтерактивного режиму

В інтерактивному режимі Python можна використовувати як калькулятор для різноманітних обчислень, а якщо додатково підключити необхідні математичні бібліотеки, то за своїми можливостями він стає практично рівним таким пакетам, як Matlab, Octave і т. п.

Наприклад, якщо ввести вираз $2+3$ і натиснути кнопку Enter, то ви отримаєте число 5, яке є результатом обчислення. Зверніть увагу, що інтерпретатор Python видає результат роботи введеної команди негайно.

```
>>>2+3
```

```
5
```

Якщо ви використовуєте IDLE або оболонку GNU/Linux чи BSD, для виходу з командного рядка інтерпретатора необхідно натиснути комбінацію кнопок Ctrl+D або виконати команду exit() (не забудьте написати дужки, "()"). Якщо ж використовуєте командний рядок Windows, для виходу з командного рядка інтерпретатора необхідно натиснути комбінацію кнопок Ctrl+Z, а потім кнопку Enter.

Пакетний режим роботи

Набирати програму в командному рядку інтерпретатора кожний раз за необхідності її виконання не досить зручно. Краще зберегти програму в файл, щоб потім, за необхідності, мати можливість запустити її.

Перш ніж приступити до написання та збереження програми мовою Python, необхідний редактор для роботи з такими файлами. Вибір редактора є вкрай важливим, використання зручного редактора допоможе зробити розробку програм більш комфортною. Досить важливим для такого редактора можна вважати інтерактивні підказки, підсвічування синтаксису та автоматизацію вставляння відступів при написанні вкладених інструкцій.

Редактор для програм мовою Python можна обрати серед великої кількості текстових редакторів (наприклад: Notepad++, Sublime Text, Komodo Edit, Vim, Emacs), спеціалізованих оболонок (наприклад: Thonny, PyCharm, Wing IDE) або використовувати вбудований редактор IDLE Python. Не рекомендується використовувати Блокнот Windows, оскільки він не має функції підсвічування синтаксису, а також не дозволяє автоматично вставляти відступи, що дуже важливо для програм, які пишуться мовою Python.

Напишемо нашу першу програму, як редактор для її написання будемо використовувати редактор, вбудований в оболонку IDLE Python.

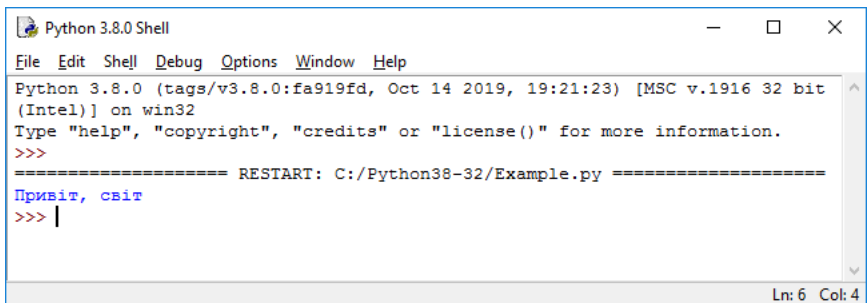
В запусненій оболонці IDLE Python оберіть послугу головного меню File/New File і в відкритому вікні введіть команду:

```
print('Привіт, Світ!')
```

Ця команда буде виводити повідомлення 'Привіт, Світ!' на екран (детально функція print() буде розглянута пізніше).

Щоб зберегти програму, скористайтеся послугою головного меню File/Save та вкажіть ім'я файлу, в якому буде збережено програму, наприклад hello.py.

Для запуску програми на виконання необхідно скористатися послугою головного меню Run/Run Module, або натиснути функціональну кнопку F5. В результаті буде активовано оболонку IDLE Python, в якій і виведеться результат роботи програми (Рис. 2.3).



```
Python 3.8.0 Shell
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python38-32/Example.py =====
Привіт, світ
>>> |
```

Рис. 2.3.

Якщо необхідно запустити програму, збережену в файлі, з терміналу або командного рядка потрібно в командному рядку викликати інтерпретатор Python і в якості аргументу передати ім'я файлу з програмою. Наприклад, для файлу з ім'ям hello.py команда запуску буде виглядати так: python hello.py або python3 hello.py. Зрозуміло, що поточним каталогом має бути каталог, в якому знаходиться файл, програми інакше необхідно вказати шлях до файлу з програмою.

В ОС Windows файл з Python-програмою (файл з розширенням .py) можна запустити як звичайний виконуваний файл. Тобто знайшовши файл на диску, запустити його подвійним клацанням лівої кнопки миші.

2.3. Змінні та літерали

Ідентифікатор – це ім'я деякої сутності (змінної, функції, класу) в програмі для її позначення. При виборі ідентифікаторів необхідно дотримуватися таких правил:

- Першим символом ідентифікатора може бути буква з алфавіту (символ ASCII в верхньому або нижньому регістрі, або символ Unicode) або символ підкреслення «_».
- Інша частина ідентифікатора (всі символи крім першого) може складатися з букв (символи ASCII в верхньому або нижньому регістрі, а також символи Unicode), символу підкреслення «_» або цифр (0-9).
- Імена ідентифікаторів чутливі до регістру. Наприклад, `myname` і `myName` - це два різні ідентифікатори (зверніть увагу на регістр літер "n" та "N").
- Ідентифікатор не може співпадати з ключовими (зарезервованими) словом інтерпретатора Python. Перелік ключових слів можна отримати, виконавши послідовно дві команди: `import keyword` та `print(keyword.kwlist)`. Для версії 3.7.1 це: 'False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield'.

Приклади допустимих імен ідентифікаторів: `a`, `__my_name`, `name_23`, `alb2_c3` і будь_які_символи_utf8_δξηϑϣϞϠ.

Приклади неприпустимих імен ідентифікаторів: `2things`, тут є пропуск, `my-name` і "це_в_кавичках".

Змінні.

При написанні програм досить часто необхідно зберігати різні дані та мати можливість маніпулювання цими даними. Ось тут як раз і знадобляться змінні. Слово «змінні» говорить сама за себе - їх значення може змінюватися, а значить, можна зберігати в змінній все, що завгодно. **Змінна в мові Python** – це просто посилання на область пам'яті комп'ютера, в якій зберігається деякі дані (посилання на деякий об'єкт). Для задання імен змінних

використовуються ідентифікатори. В змінних можуть зберігати значення різних типів.

Літерали.

Літерал (literal – константа) – постійне значення певного типу даних, записане у вихідному коді комп'ютерної програми. Прикладом літералу може бути число (наприклад: 5, 1.23, 9.25e-3) або рядок (наприклад, 'Це рядок', "It's a string!"). Значення літералу використовується буквально, число 2 завжди представляє саме себе і нічого іншого - це «константа», тому що її значення не можна змінити. В мові Python немає окремої синтаксичної конструкції для оголошення літералів. Досить часто для вказання того, що змінну не варто змінювати, її пишуть великими літерами (це вказівка розробнику, а не інтерпретатору).

2.4. Типи даних

Мова Python відноситься до мов з неявною строгою динамічною типізацією.

Неявна типізація означає, що при оголошенні змінної її тип не вказується (при явній типізації тип змінної вказується обов'язково). Як приклад мов з явною типізацією можна привести Pascal, C++. Ось як буде виглядати оголошення цілочисельної змінної у мовах Pascal, C++ та Python.

Pascal:

```
Var a: Integer;
```

C++:

```
int a=1;
```

Python:

```
a=1
```

Для мов з **динамічною типізацією** тип змінної визначається безпосередньо при виконанні програми. Окрім того, можна зазначити:

- будь-яка змінна є посиланням;
- типом змінної є те, на що вона посилається;
- тип змінної може довільно змінюватися по ходу виконання коду, коли змінна починає посилатися на інший об'єкт.

Тому замість «присвоєння значення змінної» краще говорити про «зв'язування значення з деяким ім'ям». З іншого боку, якщо мова має статичну типізацію, то тип змінної визначається на етапі компіляції і не може бути змінений на всьому протязі свого життєвого циклу.

Строга типізація (сильна типізація або strong typing) не дає можливості проводити операції у виразах з даними різних несумісних типів. Не строга типізація (слабка типізація або weakly typed) дає можливість виконати операції над несумісними типами даних і отримати деякий непередбачуваний результат. Тобто, у мовах з строгою типізацією ви не можете скласти наприклад рядки і числа, необхідне приведення до одного типу. Строго типізованими мовами є: Pascal та Python, до мов з не строгою типізацією відносяться - C і C++.

2.4.1 Вбудовані типи даних

Тип даних – це множина значень і операцій на цими значеннями.

В Python типи даних можна розділити на вбудовані в інтерпретатор (built-in) і невбудовані, які можна використовувати при імпортуванні відповідних модулів.

До основних вбудованих типів відносяться:

1. None (невизначене значення змінної)
2. Логічний тип (Boolean Type)
3. Числа (Numeric Type)
 - a. int – ціле число
 - b. float – число з плаваючою точкою (дійсне число)
 - c. complex – комплексне число
4. Послідовності (Sequence Type)
 - a. list – список
 - b. tuple – кортеж
 - c. range – діапазон
5. Рядки (Text Sequence Type)
 - a. str
6. Бінарні послідовності (Binary Sequence Types)
 - a. bytes – байти
 - b. bytearray – масиви байт

c. memoryview – спеціальні об'єкти для доступу до внутрішніх даних об'єкта через protocol buffer

7. Множини (Set Types)

- a. set – множина
- b. frozenset – незмінювана множина

8. Словники (Mapping Types)

- a. dict – словник

2.4.2. Ініціалізація змінних

Враховуючи неявну типізацію мови Python при оголошенні змінної, їй повинно бути надане значення (вона має бути ініціалізована). Щоб оголосити та ініціалізувати змінну? необхідно написати її ім'я, потім поставити оператор присвоєння (знак рівності «=») і вказати значення, з яким дана змінна буде створена. Наприклад: $z=5$.

При ініціалізації змінної на рівні інтерпретатора відбувається наступне:

- створюється цілочисельний об'єкт 5 (можна представити, що в цей момент виділяється комірка пам'яті певного об'єму і в неї записується число 5);
- за оператором «=» відбувається зв'язування змінної z і цілочисельного об'єкту 5 (визначається адреса отриманого об'єкту 5, яка присвоюється змінній z , тобто в результаті змінна z посилається на об'єкт 5).

Якщо розглянути більш складний вираз, наприклад $z=5+26.0$. Інтерпретатор обчислює значення виразу справа ($5+26.0$) і створює об'єкт з отриманим значенням (31.0). Далі аналогічно до попереднього.

Як було сказано раніше, Python є об'єктно-орієнтовною мовою програмування, тому об'єктів в мові Python багато. Спробуємо розібратися, як пов'язане поняття об'єкту з змінними та їх значеннями (при ініціалізації змінної спочатку створюється цілочисельний об'єкт). Об'єкт в даному випадку – це абстракція для подання даних певного типу, що розміщуються в пам'яті за адресою. Кожен такий об'єкт має три атрибути: адресу (ідентифікатор об'єкту), значення та тип. Ідентифікатор об'єкту – це

унікальна ознака об'єкту (адреса об'єкту в пам'яті), за якою можна відрізнити один об'єкти від іншого. Ідентифікатор представляється в вигляді деякого цілочисельного значення.

Нехай ініціалізовано змінну Z.

```
>>> z=134
```

Для отримання значення змінної достатньо ввести її ім'я.

```
>>> z
```

```
134
```

Щоб переглянути ідентифікатор об'єкту, на який посилається певна змінна, призначена функція `id()`.

```
>>> id(z)
```

```
505998272
```

Для визначення типу призначена функція `type()`.

```
>>> type(z)
```

```
<class 'int'>
```

Цікавий ефект буде при використанні цілих чисел в діапазоні від -5 до 256. Ідентифікатори об'єктів різних змінних з однаковими значеннями в межах зазначеного діапазону завжди будуть однаковими, оскільки для покращення швидкодії інтерпретатор при запуску створює об'єкти з цими значеннями.

```
>>> z=123
```

```
>>> y=123
```

```
>>> id(z)
```

```
505998096
```

```
>>> id(y)
```

```
505998096
```

Також при ініціалізації змінних можливі наступні вирази:

```
>>> x, y, z=2, 3, 4
```

```
>>> a=b=c=2
```

В першому випадку змінним `x`, `y`, `z` будуть надані значення відповідно 2, 3, 4. В другому випадку змінним `a`, `b`, `c` буде надане значення 2.

Для прикладу розглянемо задачу. «Дано дві змінні `x` та `y` з певним значенням. Написати програму, за якою змінна `x` отримає значення змінної `y`, а змінна `y` отримає значення змінної `x`». Зважаючи на зазначений принцип

роботи інтерпретатора при виконанні присвоєння і можливі вирази для задання значень змінних, дана задача розв'язується застосуванням досить простого виразу: `x, y=y, x`.

```
>>> x=500
>>> y=600
>>> x, y=y, x
>>> x
600
>>> y
500
```

2.4.3. Змінювані і незмінювані типи

Всі типи даних в Python належать до однієї з 2-х категорій: змінювані (mutable) та незмінювані (immutable).

До незмінюваних (immutable) типів належать:

- логічні (bool);
- цілі числа (int);
- числа з плаваючою точкою (float);
- комплексні числа (complex);
- кортежи (tuple);
- рядки (str);
- незмінювані множини (frozen set).

До змінюваних (mutable) типів відносяться:

- списки (list);
- множини (set);
- словники (dict).

Незмінюваність типу даних означає, що створений об'єкт цього типу більше не змінюється (не мутується). При зміні значення змінної (наданні нового значення змінній) створюється новий об'єкт і змінна посилається на нього. В цьому і полягає незмінюваність об'єкту – це не константність, оскільки значення можна змінити, проте це буде вже посилання на інший об'єкт з новим значенням.

Наприклад, якщо ми виконуємо вираз `a = 5`, то буде створено об'єкт з цілочисельним значенням 5 та унікальним ідентифікатором. В загальному можна зазначити наступне:

- Об'єкт за адресою `id1` має тип `int` і значення 5;
- Змінна `a` містить адресу `id1` (змінна `a` посилається на значення 5).



Рис. 2.4. Виконання виразу `a=5`.

```
>>> a = 5
>>> id(a)
1672501744
```

Тобто об'єкт з `id=1672501744` буде мати значення 5. Оскільки це цілочисельний тип, який є незмінюваним, то змінити значення цього об'єкту не можна. Проте це не забороняє виконати вираз `a = 6` і надати нового значення змінній `a`.

```
>>> a = 6
>>> id(a)
1873521735
```

Тобто створено новий об'єкт з новим ідентифікатором.

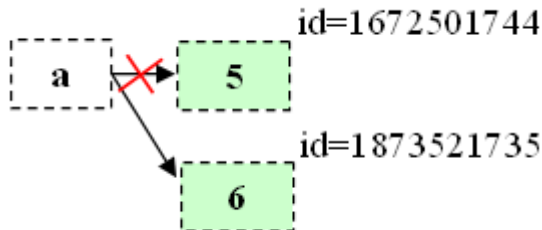


Рис. 2.5. Виконання виразу `a=6`.

2.5. Введення виведення даних

Виведення результатів роботи програми.

Для виведення результату роботи програми може бути використана функція `print()`, яка має наступний формат:

```
print(value1[, value2, ...][, sep=' '][, end='\n']  
      [, file=sys.stdout][, flush=False])
```

Використовуючи функцію `print()` можна вивести як одне, так і декілька значень. Для виведення декількох значень їх необхідно відділити один від одного комами (`value1[, value2, ...]`). Значення, що призначені для виведення, можуть бути різного типу, проте при виведенні все рівно вони будуть перетворені в рядковий тип.

Якщо виводиться декілька значень, то при виведенні одне від одного вони будуть відокремлені літералом (символом чи декількома символами), вказаним в параметрі `sep` (за замовчуванням це пропуск).

Після виведення всіх значень кінець рядку буде доповнений літералом (символом чи декількома символами), вказаними в параметрі `end` (за замовчуванням символ закінчення рядка та переходу на новий).

За замовчуванням, функція `print()` виводить значення в стандартний пристрій виведення `sys.stdout` (екран). За допомогою параметру `file` можна перенаправити виведення в інше місце – наприклад, до файлу. При цьому, якщо параметр `flush` має значення `False`, значення, що виводяться, примусово буде записане до файлу (відбувається автоматичне виштовхування з буферу). Перенаправлення виведення буде розглянуто при розгляді роботи з файлами.

Наприклад:

```
print(5 + 10)  
15  
print('Рядок1')  
print('Рядок2')  
Рядок1  
Рядок2  
print('Рядок1', 'Рядок2')  
Рядок1 Рядок2
```

```
print('Рядок1', 'Рядок2', sep='-')
Рядок1-Рядок2
print('Рядок1', 'Рядок2', end='-')
print('Рядок3')
Рядок1 Рядок2-Рядок3
```

Введення даних до програми.

При написанні програм досить часто виникає необхідність в отриманні даних для опрацювання від користувача. Для реалізації такої можливості код програми повинен містити оператори, за якими можна отримати дані, введені користувачем. В мові Python для цього передбачено функцію `input([prompt])`, за якою можна отримати дані зі стандартного пристрою введення `sys.stdin` (клавіатура).

Необов'язковий параметр `prompt`, призначений для вказання запрошення до введення, та буде виведений на стандартний пристрій виведення. Використання функції має формат:

```
[<змінна>]=input([<prompt >])
```

Для прикладу розглянемо програму, за якою буде запитуватися ім'я користувача і виводитися привітання.

```
print('Як Вас звати?')
name = input()
print('Вітаю', name)
```

Розглянемо ще одну задачу: «Написати програму обчислення суми двох чисел, заданих користувачем». Для розв'язання задачі необхідно: вивести запит користувачу на введення чисел, зчитати два числа, введені користувачем, виконати операцію додавання та результат вивести на екран.

Базуючись на раніше викладеному матеріалі та елементарних знаннях з математики, створимо програму такого вигляду:

```
print('Введіть два числа:')
a = input()
b = input()
s = a + b
print(s)
```

Проте після запуску програми і введення значень, наприклад 2 та 3, в результаті виконання програми отримаємо 23, хоча напевне нами очікувалося 5. Це пов'язане з тим, що за функцією `input()` повертається значення рядкового типу. А вказана операція «+» для змінних рядкового типу виконує їх об'єднання (детальна робота з рядковими змінними буде розглянута пізніше).

Отже необхідно отримані від функції `input()` рядкові значення перетворити в цілі або дійсні числа. Для цього можна скористатися функціями `int()` або `float()` відповідно.

Тоді програма матиме вигляд:

```
print('введіть два числа')
a = int(input())
b = int(input())
s = a + b
print(s)
```

3. Числові дані

Як було сказано раніше, в Python існує три вбудованих числових типи даних:

- цілі числа. Наприклад: 2, 3, 5. Цілі числа можуть бути довільної довжини, тобто у мові Python з цілими числами працюють в режимі довгої арифметики (можна обчислити значення 2^{2019});
- дійсні числа. Наприклад: 3.23, 52.3E-4 (E вказує степінь числа 10, в даному випадку $52.3 \cdot 10^{-4}$);
- комплексні числа. Наприклад: $(-5+4j)$, $(2.3 - 4.6j)$.

3.1. Робота з цілими та дійсними числами

3.1.1. Математичні та бітові операції

Математичні операції

Символ операції	Призначення	Використання	Приклад
+	Додавання	$x + y$	2 + 3 результат 5 При a=3 та b=2 a + b результат 5
-	Віднімання	$x - y$	3 - 2 результат 1
*	Множення	$x * y$	2 * 3 результат 6
/	Ділення	x / y	4 / 2 результат 2.0 5 / 2 результат 2.5
//	Знаходження цілої частини від цілочисельного ділення	$x // y$	7 // 3 результат 2 7 // -3 результат -3 -7 // 3 результат -3 -7 // -3 результат 2
%	Знаходження залишку від цілочисельного ділення	$x \% y$	7 % 3 результат 1 7 % -3 результат -2 -7 % 3 результат 2 -7 % -3 результат -1 <i>Пояснення:</i> $c = a // b$ $z = a \% b$, z має той же знак що і b таким чином, щоб $a = c * b + z$
**	Піднесення до степеня	$x ** y$	2**3 результат 8 4**0.5 результат 2.0

Якщо в якості операндів деякого арифметичного виразу використовуються тільки цілі числа, то результат теж буде ціле число, винятком є операція ділення, результатом якої буде дійсне число. При спільному використанні цілих і дійсних чисел результатом буде дійсне число.

Бітові операції над цілими числами

В мові Python передбачені бітові (побітові) операції над цілими числами. Такі операції називаються бітовими, тому що виконуються послідовно над окремими бітами операндів (побітово).

Символ операції	Призначення	Використання	Приклад
&	Бітове «І» (AND)	$x \& y$	9&3 результат 1 Пояснення: $9_{10}=1001_2$ $3_{10}=0011_2$ Після бітового «AND» отримаємо: $0001_2=1_{10}$
	Бітове «АБО» (OR)	$x y$	9 3 результат 11 Пояснення: $9_{10}=1001_2$ $3_{10}=0011_2$ Після бітового «OR» отримаємо: $1011_2=11_{10}$
^	Бітове ВИКЛЮЧАЮЧЕ АБО (XOR)	$x \wedge y$	9 ^ 3 результат 10 Пояснення: $9_{10}=1001_2$ $3_{10}=0011_2$ Після бітового «XOR» отримаємо: $1010_2=10_{10}$
~	Бітова операція НІ (NOT). Для числа x відповідає $-(x+1)$	$\sim x$	~ 9 результат -10
<<	Зсув вправо на кількість біт. До числа в двійковому записі справа дописується вказана кількість нулів	$x \ll n$	9 << 1 результат 18 Пояснення $9_{10}=1001_2$ Після дописування справа одного 0 отримаємо: $10010_2=18_{10}$
>>	Зсув вліво. З числа в двійковому записі забирається вказана кількість розрядів	$x \gg n$	9 >> 1 результат 4 Пояснення $9_{10}=1001_2$ Після прибирання справа одного символу отримаємо: $100_2=4_{10}$

Короткий запис математичних та бітових операцій

Досить часто результат проведення певної математичної або бітової операції необхідно присвоїти змінній, над якою ця операція проводилася. Тобто необхідно модифікувати (змінити значення) змінну, нове значення якої буде залежати від її попереднього значення. Для цього в мові Python існують короткі форми запису виразів, тобто вираз виду «змінна = змінна операція вираз» прийме вигляд «змінна операція = вираз».

Так вираз: $a = a + 3$, перепишеться в вигляді $a += 3$.

Така коротка форма може бути застосовна до всіх зазначених вище математичних та бітових операцій.

3.1.2. Порядок обчислення операцій

Якщо маємо вираз виду $2 + 3 * 4$, то з шкільного курсу математики відомо, що спочатку виконується операція множення, а вже потім операція додавання, оскільки операція множення має вищий пріоритет, ніж операція додавання.

Так само і в операціях мови Python, спершу обчислюються оператори і вирази з вищим пріоритетом, а потім поступово за спаданням пріоритету.

В таблиці наведено пріоритет операторів мови Python, починаючи з самого низького (зверху таблиці) і до найвищого (внизу таблиці).

Оператор	Опис
lambda	Лямбда-вираз (Лямбда-функція)
or	Логічне «АБО»
and	Логічне «І»
not x	Логічне «НІ»
in, not in	Перевірка приналежності
is, is not	Перевірка тотожності
<, <=, >, >=, !=, ==	Оператори порівняння
	Бітове «АБО»
^	Бітове «ВИКЛЮЧАЮЧЕ АБО»
&	Бітове «І»
<<, >>	Зсуви
+, -	Додавання та віднімання
*, /, //, %	Множення, ділення, цілочисельне ділення та залишок від ділення
+x, -x	Додатне, від'ємне
~x	Бітове «НІ»

**	Піднесення до степеня
x.attribute	Посилання на атрибут
x[індекс]	Звернення за індексом
x[індекс1:індекс2]	Зріз
f(аргументи ...)	Виклик функції
(вираз, ...)	Кортеж (tuple)
[вираз, ...]	Список (list)
{ключ:дані, ...}	Словник (dict)

В таблиці оператори з рівним пріоритетом розміщені в одному рядку (наприклад, + та – мають рівний пріоритет).

Оператори, які ще не були розглянуті, будуть описані в наступних розділах.

Так само як і в математиці, в мові Python при формуванні математичних виразів для зміни порядку обчислень можуть використовуватися дужки. Наприклад $(2 + 3) * 4$.

3.1.3. Вбудовані функції цілих і дійсних чисел

В мові Python поряд з вбудованими типами є вбудовані функції, які містяться в стандартній бібліотеці і доступні без будь-яких додаткових вказівок. Розглянемо функції, що можуть бути застосовані до цілих та дійсних чисел.

abs (X) – повертає абсолютне значення (модуль) числа.

divmod (A, B) – повертає пару чисел (P, R), які є цілою частиною P та остачею R при виконанні цілочисельного ділення. Для цілих чисел результат буде таким самим, як і при $(A // B, A \% B)$. Для дійсних чисел результатом є $(Q, A \% B)$, де Q зазвичай $\text{math.floor}(A / B)$, але може бути і на 1 менше. Незважаючи на це значення за виразом $Q * B + A \% B$ дуже близьке до A, якщо $A \% B$ не рівне нулю, то має такий самий знак, як і B, $0 \leq \text{abs}(A \% B) < \text{abs}(B)$.

```
>>> divmod(7, 2)
(3, 1)
```

pow (X, Y[, Z]) – повертає X в степені Y за модулем Z (обчислюється більш ефективно, чим $\text{pow}(X, Y) \% Z$). Двоаргументна форма $\text{pow}(X, Y)$ – еквівалент використання оператора піднесення до

степеня: $X**Y$. Якщо параметр Z заданий, то X та Y мають бути цілочисельними, окрім того Y має бути невід'ємним.

```
>>> pow(2, 3)
8
>>> pow(2, 3, 3)
2
```

round(number[, ndigits]) – повертає число `number`, округлене до `ndigits` знаків після десяткової точки.

Проте поведінка `round()` для дійсних чисел може бути несподіваною, це результат факту, що деякі десяткові дроби не можуть бути представлені точно як дійсні числа.

```
>>> round(2.65, 1)
2.6
>>> round(2.75, 1)
2.8
>>> round(2.85, 1)
2.9
>>> round(2.665, 2)
2.67
>>> round(2.675, 2)
2.67
```

Якщо параметр `ndigits` не заданий, то округлення відбувається до найближчого цілого числа. Проте, якщо дробова частина = 0.5, то виконується «Банківське округлення», тобто округлення до найближчого парного числа.

```
>>> round(2.5, 2)
2
>>> round(3.5, 2)
4
```

max(arg1, arg2, *args, *[, key=func]) – повертає найбільше значення з двох чи більше аргументів. Ця функція має більш широкі можливості, але про них пізніше.

min(arg1, arg2, *args, *, key=func) – повертає найменше значення з двох чи більше аргументів. Ця функція має більш широкі можливості, але про них пізніше.

int([object], [osn]) – повертає перетворене значення object до цілого десяткового числа. osn визначає систему числення задання object (osn від 2 до 36 включно). За замовчуванням object=0, osn=10.

```
>>> int(4.9)
4
>>> int('11')
11
>>> int('11',2)
3
```

float([X]) – повертає перетворене X до дійсного числа.

```
>>> float('1.23')
1.23
>>> float('1e-003')
0.001
>>> float(3)
3.0
```

bin(X) – повертає рядковий запис цілого числа X в двійковій формі.

```
>>> bin(5)
'0b101'
```

hex(X) – повертає рядковий запис цілого числа X в шістнадцятковій формі. Для перетворення дійсного числа використовується метод float.hex(X).

```
>>> hex(255)
'0xff'
>>> float.hex(3.4)
'0x1.b3333333333333p+1'
```

oct(X) – повертає рядковий запис цілого числа X у вісімковій формі.

```
>>> oct(12)
'0o14'
```

bool([X]) – повертає приведені значення X до логічного типу (bool), використовуючи стандартну процедуру перевірки істинності. Повертає логічне значення True або False.

```
>>> bool(1)
True
>>> bool(15)
True
>>> bool(0)
False
```

3.1.4. Модуль math

Окрім стандартної бібліотеки з досить великим набором функцій, мова Python містить велику кількість додаткових бібліотек, які можуть бути використані при написанні програм. Так однією з бібліотек, яка містить математичні функції і призначена для роботи з числовими даними, є бібліотека (модуль) math.

Для роботи з цим модулем його попередньо необхідно імпортувати (підключити), виконавши команду `import math`. Більш детально про імпорт та під'єднання модулів подано в додатку 1.

Розглянемо константи та функції, що містяться в бібліотеці math.

Константи:

math.pi. Число π (`math.pi` $\approx 3.141592653589793$).

math.e. Число e (`math.e` $\approx 2.718281828459045$).

math.tau. Математична константа кола, рівна 2π .

math.inf. Додатна нескінченність (дійсне значення). Еквівалентно результату `float('inf')`. Для від'ємної нескінченності використовується `-math.inf`.

math.nan. "не число" ("not a number" (NaN)). Еквівалент результату `float('nan')`.

Перевірка значень:

math.isclose(A, B, *, rel_tol=1e-09, abs_tol=0.0).

Повертає True, якщо дійсні числа A та B близькі одне до одного, і False в

іншому випадку. Чи є числа близькими, визначається в відповідності з даними абсолютним і відносним відхиленням.

`rel_tol` (відносне відхилення) - це максимальна допустима різниця між `A` та `B`, відносно більшої абсолютної величини `A` або `B`. Наприклад, щоб встановити відхилення в 5%, вказується `rel_tol=0.05`. Відхилення за замовчуванням дорівнює `1e-09`, це гарантує, що два значення однакові приблизно в 9 десяткових цифрах. `rel_tol` має бути більше нуля.

`abs_tol` (абсолютне відхилення) - це мінімальне абсолютне відхилення. Використовується для порівнянь значень близьких до нуля. `abs_tol` має бути більше або рівне нуля.

Логічний вираз функції можна представити в вигляді:

```
abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
```

Значення `NaN` не вважається близьким до будь-якого іншого значення (навіть до себе). Значення `math.inf` та `-math.inf` вважаються близькими лише до себе.

`math.isfinite(x)`. Повертає `True`, якщо `x` не є ні нескінченністю (`math.inf` та `-math.inf`) ні `NaN`, і `False` в іншому випадку. Зауважимо, що `0.0` вважається скінченним.

`math.isinf(x)`. Повертає `True`, якщо `x` є нескінченністю, і `False` в іншому випадку.

`math.isnan(x)`. Повертає `True`, якщо `x` є `NaN`, і `False` в іншому випадку.

Функції округлення чисел:

`math.ceil(x)`. Повертає найближче ціле число, більше чим `x` (округлення вгору).

```
>>> math.ceil(3.1)
```

```
4
```

```
>>> math.ceil(-3.1)
```

```
-3
```

`math.floor(x)`. Повертає найближче ціле число, менше чим `x` (округлення вниз).

```
>>> math.floor(3.8)
```

```
3
```

```
>>> math.floor(-3.8)
-4
```

math.trunc(X). Повертає усічене значення X до цілого.

```
>>> math.ceil(3.8)
4
```

```
>>> math.ceil(-3.8)
-3
```

Функції представлення та теоретико-числові функції:

math.copysign(X, Y). Повертає дійсне число, що має абсолютне значення таке ж, як і у числа X, а знак - як у числа Y.

```
>>> math.copysign(3, -5)
-3.0
```

```
>>> math.copysign(-3, 5)
3.0
```

math.fabs(X). Повертає абсолютне значення числа.

```
>>> math.fabs(-7.3)
7.3
```

math.frexp(X). Повертає мантису і експоненту числа у вигляді пари (M, E), таким чином, що, $X=M*2^E$. M – дійсне число ($0,5 \leq \text{abs}(M) < 1$), а E - ціле число.

```
>>> math.frexp(7.3)
(0.9125, 3)
```

math.ldexp(X, I) – повертає $X*2^I$. Функція, зворотна функції `math.frexp()`.

```
>>> math.ldexp(0.9125, 3)
7.3
```

math.modf(X). Повертає дробову і цілу частину числа X у вигляді пари (D, C). Обидва числа є дійсними і мають той же знак, що і X.

```
>>> math.modf(5)
(0.0, 5.0)
```

```
>>> math.modf(4.5)
(0.5, 4.0)
```

```
>>> math.modf(-4.5)
(-0.5, -4.0)
```

math.fmod(X, Y). Повертає залишок від ділення X на Y . Для додатних чисел аналогічно оператору $X \% Y$. Якщо серед чисел X , Y є від'ємні, то залишок обчислюється для абсолютних величин чисел X та Y , а результат матиме такий же знак, що і X . Окрім того, більш точно працює з дійсними числами.

```
>>> math.fmod(7.5, 3)
1.5
>>> math.fmod(-7.5, 3)
-1.5
>>> math.fmod(7.5, -3)
1.5
>>> math.fmod(-7.5, -3)
-1.5
```

math.factorial(X). Повертає факторіал числа X .

```
>>> math.factorial(5)
120
```

math.fsum(iterable). Повертає суму всіх членів з `iterable`. Аналог вбудованої функції `sum()`, але `math.fsum()` більш точна для дійсних чисел.

```
>>>sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>>math.fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1,
.1])
1.0
```

math.gcd(A, B). Повертає найбільший спільний дільник цілих чисел A та B (найбільше натуральне число, на яке ці числа діляться без остачі). `math.gcd(0, 0)` повертає 0.

```
>>>math.gcd(6, 8)
2
>>>math.gcd(5, 7)
1
```


Степеневі та логарифмічні функції:

math.exp (X). Повертає значення e^X . Аналогічно до `math.e**X`.

math.expm1 (X). Повертає значення $e^X - 1$. При $X \rightarrow 0$ точніше, ніж `math.exp(X) - 1`.

math.log (X, [base]). Повертає значення логарифму числа X за основою `base`. Якщо `base` не вказано, обчислюється значення натурального логарифму.

```
>>> math.log(8)
2.0794415416798357
>>> math.log(8, 2)
3.0
```

math.log10 (X). Повертає значення логарифму числа X за основою 10. Аналогічно до `log(x, 10)`.

```
>>> math.log10(8)
0.9030899869919435
```

math.log2 (X). Повертає значення логарифму числа X за основою 2. Аналогічно до `log(x, 2)`.

```
>>> math.log2(8)
3.0
```

math.log1p (X). Повертає значення натурального логарифму для $(1 + X)$. При $X \rightarrow 0$ точніше, ніж `math.log(1 + X)`.

math.pow (X, Y). Повертає дійсне значення X^Y . Для `pow(1.0, A)` та `pow(A, 0.0)` завжди повертається 1.0, навіть якщо A рівне нулю або NaN. Якщо X та Y скінченні, X від'ємне, а Y не є цілим числом, тоді результат `pow(X, Y)` невизначений.

```
>>> math.pow(3, 2)
9.0
>>> math.pow(-3, 3)
-27.0
```

math.sqrt (X). Повертає значення квадратного кореня числа X .

Функції кутових перетворень:

math.degrees (X). Конвертує радіани в градуси.

```
>>> math.degrees(math.pi/4)
```

45.0

math.radians (X). Конвертує градуси в радіани.

```
>>> math.radians(45)
```

```
0.7853981633974483
```

Тригонометричні функції:

math.cos (X). Повертає значення косинусу числа (X вказується в радіанах).

math.sin (X). Повертає значення синусу числа (X вказується в радіанах).

math.tan (X). Повертає значення тангенсу числа (X вказується в радіанах).

math.acos (X). Повертає значення арккосинусу числа в радіанах.

math.asin (X). Повертає значення арксинусу числа в радіанах.

math.atan (X). Повертає значення арктангенсу числа в радіанах.

math.atan2 (Y, X). Повертає значення арктангенсу Y/X в радіанах.

З урахуванням чверті, в якій знаходиться точка (X, Y), тобто результат в межах від $-\pi$ до π .

```
>>> math.degrees(math.atan2(1, 1))
```

```
45.0
```

```
>>> math.degrees(math.atan2(-1, 1))
```

```
-45.0
```

```
>>> math.degrees(math.atan2(-1, -1))
```

```
-135.0
```

```
>>> math.degrees(math.atan2(1, -1))
```

```
135.0
```

math.hypot (X, Y). Повертає значення гіпотенузи трикутника з катетами X та Y (аналогічно до $\text{math.sqrt}(x * x + y * y)$).

Гіперболічні функції:

math.cosh (X). Повертає значення гіперболічного косинусу числа.

math.sinh (X). Повертає значення гіперболічного синусу числа.

math.tanh (X). Повертає значення гіперболічного тангенсу числа.

math.acosh(x). Повертає значення оберненого гіперболічного косинусу числа.

math.asinh(x). Повертає значення оберненого гіперболічного синусу числа.

math.atanh(x). Повертає значення оберненого гіперболічного тангенсу числа.

Спеціальні функції:

math.erf(x). Повертає значення функції помилок для x . Функція помилок - це неелементарна функція, що використовується в теорії ймовірності, статистиці і математичній фізиці. Вона визначається як:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

math.erfc(x). Повертає значення додаткової функції помилок для x (аналогічно до $1 - \text{math.erf}(x)$).

math.gamma(x). Повертає значення гамма-функції для x . Гамма-функція визначається формулою: $\Gamma(x) = \int_0^{\infty} s^{x-1} e^{-s} ds = \int_0^1 \left(\ln \frac{1}{s}\right)^{x-1} ds$.

math.lgamma(x). Повертає значення натурального логарифму гамма-функції для x .

Більш детально з функціями модуля `math` можна ознайомитися на сторінці офіційної документації:

<https://docs.python.org/3/library/math.html?highlight=math#angular-conversion>

3.2. Приклади розв'язування задач

Приклад. Написати програму для розрахунку ідеальної ваги чоловіка та жінки за формулою Брокка.

Ідеальна вага для чоловіка = (зріст в сантиметрах - 100) * 1,15.

Ідеальна вага для жінки = (зріст в сантиметрах - 110) * 1,15.

```
zrist=int(input('Ваш зріст в сантиметрах:'))
v_men=(zrist-100)*1.15
v_women=(zrist-110)*1.15
```

```
print('Ідеальна вага для чоловіка = ', v_men)
print('Ідеальна вага для жінки = ', v_women)
```

Приклад. Написати програму обчислення значення виразу

$$e^{x+y} + \frac{5}{\cos(y-x) + 3}$$

```
import math
x=float(input('X= '))
y=float(input('Y= '))
res=math.exp(x+y)+5/(math.cos(y-x)+3)
print('Знаення виразу = ', res)
```

3.3. Робота з комплексними числами

Для задання комплексного числа можна використати запис $A + Bj$.

```
>>>z=1+2j
>>>print(z)
(1+2j)
```

Інший спосіб задання комплексного числа – це використання функції `complex([real[, imag]])`, за якою формується число з значенням `real + imag*j`. Якщо перший параметр є рядком (другий параметр має бути відсутній), то він буде вважатися рядковим записом комплексного числа, і за функцією `complex()` буде сформоване комплексне число, яке відповідатиме даному рядковому значенню. При перетворенні із рядка, він не має містити пропусків навколо центрального оператора `+` або `-`.

```
>>>x=complex(3, 2)
>>>x
(3+2j)
>>>complex('3-2j')
(3+2j)
```

Над комплексними числами можна виконувати операції: додавання, віднімання, множення, ділення і піднесення до степеня.

```
>>> x+z
(4+4j)
>>> x-z
```

```

(2+0j)
>>> x*z
(-1+8j)
>>> x/z
(1.4-0.8j)
>>> x**z
(-1.1122722036363393-0.012635185355335208j)
>>> x**3
(-9+46j)

```

З комплексного числа можна виділити дійсну та уявну частини.

```

>>> z.real
1.0
>>> z.imag
2.0

```

Для виконання математичних операцій з комплексними числами може бути використана бібліотека `cmath`.

4. Винятки та їх обробка

Виняток (exception) – це аварійний стан, який відбувається в кодовій послідовності під час виконання програми. Прикладом є — ділення на нуль, помилки читання з файлу, вичерпання доступної пам’яті тощо. Іншими словами – це помилки, які можуть виникнути при виконанні програми. В ряді мов програмування необхідно заздалегідь передбачити можливість тієї чи іншої помилки і визначити шлях її обробки. В Python для цього передбачений спеціальний механізм винятків.

Про винятки в Python можна говорити як про тип даних, що містить інформацію про помилки. На рівні з поняттям виняток можна зустріти поняття виняткова ситуація, тобто випадок, коли виник виняток, проте досить часто ці два поняття використовуються як синоніми.

Винятки

Розглянемо як приклад, виняток ділення на нуль. Якщо спробувати виконати операцію, $1/0$ то виникне помилка, оскільки на 0 ділити не можна.

Інтерпретатор відреагує на цю помилку генерацією винятку (припиненням подальшого виконання програми) та виведе відповідне повідомлення.

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

Розберемо це повідомлення докладніше:

- `Traceback (most recent call last)` – повідомлення про те, що інтерпретатор «зловив» виняток;
- `File "<pyshell#0>", line 1, in <module>` – звідки було запущено код програми, в якому виник виняток (це може бути ім'я файлу, вказівка на інтерактивний режим в консолі – `stdin`, вказівка на інтерактивний режим оболонки IDLE - `pyshell#0`) і номер рядка, в якому це сталося;
- `1/0` – вираз, в якому стався виняток;
- `ZeroDivisionError: division by zero` – тип винятку (назва винятку) і його короткий опис.

Наведемо приклади деяких інших винятків:

Операція застосована до об'єкту невідповідного типу:

```
>>> 2 + '1'
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    2 + '1'
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

Функція отримує аргумент правильного типу, але некоректного значення:

```
>>> int('qwerty')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int('qwerty')
ValueError: invalid literal for int() with base 10:
'qwerty'
```

Синтаксична помилка:

```
>>> t:=3
File "<pyshell#0>", line 1, in <module>
    t:=3
      ^
SyntaxError: invalid syntax
```

У цих прикладах генеруються винятки: `TypeError`, `ValueError`, `SyntaxError`. Ієрархію вбудованих в Python винятків подано в додатку 4.

Обробка винятків

Про винятки можна ще сказати, що це помилки, які можна обробити. Обробка винятків або обробка виняткових ситуацій – механізм мови програмування, призначений для опису реакції програми на помилки часу виконання та інші можливі проблеми, які можуть виникати при виконанні програми і призвести до неможливості (безглуздості) подальшого відпрацювання програмою її базового алгоритму.

Обробка винятків може полягати у гарантованому виконанні певного коду або в корекції стану програми, що викликав виняток. Особливо гостро питання надійності постають у програмах, пов'язаних з обробкою математичних об'єктів, таких, як наприклад функції, оскільки існує проблема області визначення, наявність розривів тощо, що призводить до помилок у обчисленнях з плаваючою точкою. Якісна програма повинна обробити такі помилки, присвоївши відповідній змінній певне значення або, якщо це необхідно, повідомивши про виникнення помилки користувача.

Знаючи, в яких місцях і за яких обставин можуть виникнути винятки, ми можемо передбачити їх обробку. Для обробки винятків використовується конструкція `try-ехсепт`, яка має декілька форм. В конструкції `try-ехсепт` можуть бути присутні оператори: `try`, `ехсепт`, `else`, `finally`, `raise`. Проте про них по порядку.

Найпростіший варіант конструкції `try - ехсепт` для обробки винятку має вигляд:

```
try:
    #Код блоку try (код, в якому може виникнути
        виняток)
except Назва_винятку:
```

```
#Код блоку except (код, що виконується при  
вказаному винятку)
```

Вказана конструкція try - except виконується таким чином:

- Спочатку виконується код блоку try (вираз чи вирази між ключовими словами try і except).
- Якщо під час виконання коду блоку try не відбулося ніякого винятку, код блоку except пропускається і виконання конструкції try - except закінчено.
- Якщо під час виконання коду блоку try виникає виняток, виконання коду блоку try припиняється і управління переходить до обробника except.
- Якщо тип винятку, що виник, відповідає назві винятку, вказаного після ключового слова except, виконується код блоку except. Після завершення виконання коду блоку except виконується код, що міститься після конструкції try - except.
- Якщо виник виняток, тип якого не збігається з назвою типу винятку, вказаного після ключового слова except, виняток передається на зовнішню конструкцію try - except, якщо зовнішня конструкція відсутня чи обробник не знайдений і там, то виняток стає необробленим і виконання програми зупиняється з системним повідомленням про помилку, як показано вище.

Отже, в блоці try розміщується код, в якому може виникнути виняток, а в блоці except розміщується код для обробки відповідного винятку.

Незважаючи на те, що в приведеному варіанті конструкції try – except вказано лише єдину назву винятку, насправді буде оброблений як сам виняток, назву якого вказано, так і його нащадки. Наприклад, при перехопленні винятку ArithmeticError також будуть перехоплюватися і винятки FloatingPointError, OverflowError, ZeroDivisionError.

Якщо необхідно однаково обробити винятки, ієрархічно не пов'язані між собою, у рядку except можна перерахувати назви кількох таких винятків, взявши їх в дужки. Наприклад:

```
try:  
    #Код блоку try (код, в якому можуть виникнути  
        винятки)  
except (RuntimeError, TypeError, NameError):
```



```
#Код блоку except (код, що виконується при  
вказаних винятках)
```

Також можливе застосування ключового слова `except` без указання назв винятків. Такий обробник буде перехоплювати всі винятки (в тому числі і системні: переривання з клавіатури, системний вихід і т.д.), і тому в такій формі `except` практично не використовується. В тих випадках, коли є необхідність в обробці всіх вбудованих несистемних винятків, може бути використаний запис `except Exception`.

В тих випадках, коли необхідно опрацювати по-різному різні винятки, що можуть виникнути в певному операторі (наборів операторів), в конструкції `try - except` можна розмістити декілька відповідних блоків `except`. Тоді при виникненні винятку будуть переглянуті блоки `except` по черзі, зверху до низу, в пошуках обробника відповідного винятку (тільки перший обробник, що підходить, буде виконаний).

Виходячи з сказаного конструкцію `try – except` можна представити в наступному вигляді:

```
try:  
    #Код блоку try  
except Назва_винятку1:  
    #Код блоку, що виконується при вказаних винятках  
except (Назва_винятку2, Назва_винятку3, ...):  
    #Код блоку, що виконується при вказаних винятках  
. . .  
except Exception:  
    #Код блоку, що виконується за будь-якого не  
        системного винятку, якого не було оброблено
```

Але й на цьому етапі вказана конструкція `try – except` не є повною. Конструкція `try – except` може мати ще два блоки: `finally` та `else`. Код блоку `finally` виконується в будь-якому випадку, незалежно від того, чи виник виняток в блоці `try`, чи ні. Код блоку `else` виконується в тому випадку, якщо винятку в блоці `try` не було. Блок `else` досить добре описує частину дерева розв'язку: «Якщо цього виконати не можна, то (інакше) виконати це».

Якщо блок `else` присутній, то він має йти після всіх блоків `except`, але до блоку `finally`.

```
try:
    #Код блоку try
except Назва_винятку1:
    #Код блоку, що виконується при вказаних винятках
except (Назва_винятку2, Назва_винятку3, ...):
    #Код блоку, що виконується при вказаних винятках
    . . .
except Exception:
    #Код блоку, що виконується за будь-якого не
        системного винятку, якого не було оброблено
else:
    #Код блоку, що виконується, якщо не було винятків
finally:
    #Код блоку, що виконується в будь-якому випадку,
        можливо після відповідного блоку except
```

Для прикладу напишемо програму, в якій для введеного числа X буде знаходитися частка $1/X$. Без опрацювання винятків дана програма буде мати наступний вигляд:

```
x=int(input())
k=1/x
print(k)
```

Аналізуючи виконувані операції, можна зазначити дві з них, в яких можуть виникнути винятки:

- `x=int(input())` – якщо користувач введе не ціле число, то функція `int()` не зможе введене значення привести до цілого і виникне виняток `ValueError`.
- `k=1/x` – якщо користувач введе `0`, то інтерпретатору не вдасться виконати ділення на `0`, і виникне виняток `ZeroDivisionError`.

Доповнивши програму блоком опрацювання винятку, отримаємо:

```
try:
    x=int(input())
    k=1/x
```

```

except ValueError:
    print('Помилка: очікувалося ціле число.')
except ZeroDivisionError:
    print('Помилка: ділення на ноль.')
else:
    print(k)

```

Тобто при виникненні винятку буде виведене відповідне повідомлення про помилку.

Зв'язування винятку зі змінною.

Бувають ситуації, коли при обробці винятків достатньо виведення системного опису винятку. В такому випадку немає сенсу обробляти окремі винятки, достатньо обробити виняток Exception, але при цьому повернути опис винятку, що виник і є нащадком Exception. Для цього виняток, що виник, можна зв'язати зі змінною, використовуючи оператор as:

```

except Exception as <ім'я змінної>

```

Надалі в середині відповідного блоку except можна буде звернутися до змінної і отримати дані про виняток.

Наприклад, нашу попередню задачу перепишемо:

```

try:
    x=int(input())
    k=1/x
except Exception as mes:
    print('Помилка', type(mes),':', mes)
else:
    print(k)

```

Якщо користувач введе не ціле число (наприклад, 33.4), то отримає повідомлення:

```

Помилка <class 'ValueError'> : invalid literal for
int() with base 10: '33.4'

```

Якщо користувач введе 0, то отримає повідомлення:

```

Помилка <class 'ZeroDivisionError'> : division by
zero

```

Виклик винятків

Оператор `raise` дозволяє програмісту згенерувати вказаний виняток. В єдиному аргументі `raise` вказується виняток, який буде викликано, наприклад:

```
>>> raise ZeroDivisionError('Ділення на нуль.')
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    raise ZeroDivisionError('Ділення на нуль.')
ZeroDivisionError: Ділення на нуль
```

5. Організація розгалужень в програмах

5.1. Логічні вирази і логічний тип даних

Досить часто в реальному житті зустрічаються твердження, з якими ми погоджуємося чи ні. Наприклад, якщо вам скажуть, що сума чисел 2 та 3 більше 4, ви погодитесь і скажете: "Так, це правда". Якщо ж хтось буде стверджувати, що сума чисел 2 та 3 менше 4, то ви сприймете це твердження як хибне.

Подібні твердження допускають лише дві можливих відповіді - або "так", коли твердження оцінюється як істинне (правдиве), або "ні", коли твердження оцінюється як хибне (помилкове). Такі твердження ще називають логічними виразами або логічними твердженнями. Логічний вираз в програмуванні – це конструкція мови програмування, результатом обчислення якої є «істина» або «хиба».

Логічним (булевим) типом даних в мові Python є тип `bool`, що може набувати одного з двох значень: `True` (істина) або `False` (хиба). Проте в мові Python істинним або хибним може бути не лише логічний вираз, але і об'єкт.

- Число не рівне нулю, або непорожній об'єкт інтерпретується як істина.
- Нуль, порожні об'єкти і спеціальний об'єкт `None` інтерпретуються як хиба.

5.2. Оператори відношень (порівнянь)

В Python для порівняння об'єктів (змінних різних типів) є наступні операції порівняння:

- `>` – більше;

- < – менше;
- >= – більше або рівне (не менше);
- <= – менше або рівне (не більше);
- == – дорівнює (рівне);
- != – не дорівнює (не рівне).

Результатом операції порівняння є змінна логічного типу `bool`.

```
>>> 6>5
True
>>> 7<1
False
>>> 7==7
True
>>> 7!=7
False
```

5.3. Умовний оператор `if-else` (`if-elif-else`)

Всі раніше розглянуті програми мали лінійну структуру, тобто всі інструкції виконувалися послідовно одна за одною, і кожна записана інструкція обов'язково виконувалась. За необхідності змінити порядок виконання операторів програми в залежності від виконання певних умов, тобто здійснити розгалуження процесу обчислень, використовують умовний оператор. [7]

Наприклад, необхідно для заданого числа x визначити: воно додатне, якщо $x \geq 0$, чи від'ємне в іншому випадку. Структура програми вже не може бути лінійною, оскільки залежно від значення x ($x \geq 0$ або $x < 0$) має бути виведене одне чи інше повідомлення.

Для розв'язання цієї задачі можна скористатися умовним оператором `if-else`, який ще називається оператором розгалуження. Використовуючи вказаний оператор, можна організувати виконання тих чи інших операторів в залежності від деякого логічного виразу (умови).

Синтаксис оператора `if-else`:

```
if Логічний_вираз:
    Блок_інструкцій_1
```

```
[else:  
    Блок_інструкцій_2]
```

Під час виконання оператора if-else обчислюється значення логічного виразу. Якщо значення логічного виразу істинне (True), то виконується блок_інструкцій_1 (вирази, що є вкладеними в if). Якщо значення логічного виразу хибне (False), виконується блок_інструкцій_2 (вирази, що є вкладеними в else).

Умовний оператор може бути неповним, якщо в ньому відсутня гілка else, тобто відсутнє службове слово else з відповідним йому блоком інструкцій.

Як можна бачити, запис службового слова if з логічним виразом та службового слова else і відповідних їм блоків інструкцій оформлюються в коді, як основна інструкція та вкладений блок інструкцій. Тобто всі інструкції, які відносяться до одного вкладеного блоку, повинні мати рівну величину відступу, тобто однакове число пропусків на початку рядка. Рекомендується використовувати відступ в 4 пропуски і не рекомендується використовувати в якості відступу символ табуляції.

Приклад. Написати програму для визначення, чи є задане число додатним чи не є додатним:

```
x = int(input("Введіть число: "))  
if x > 0:  
    print("Число додатне. ")  
else:  
    print("Число не є додатне. ")
```

Розширивши умову задачі необхідності визначення, чи є число додатним, чи є число від'ємним чи число є нулем (число 0 не є додатним і не є від'ємним). Виникне необхідність доповнити програму додатковою можливістю перевірки числа на рівність нулю і виведенням відповідного повідомлення.

Це можна зробити декількома способами.

Спосіб 1. Використання окремих перевірок.

```
x = int(input("Введіть число: "))  
if x > 0:  
    print("Число додатне. ")
```

```
if x < 0:
    print("Число від'ємне. ")
if x == 0:
    print("Число нуль. ")
```

У цій програмі відбувається перевірка всіх можливих випадків (більше, менше, рівне) з виведенням відповідних повідомлень. Варто відмітити, що для перевірки 3-х взаємовиключних умов можна скористатися методом виключення третього (якщо не перше і не друге, то третє) і використати лише дві перевірки.

Перепишемо програму.

Спосіб 2. Використання вкладених перевірок.

```
x = int(input("Введіть число: "))
if x > 0:
    print("Число додатне.")
else:
    if x < 0:
        print("Число від'ємне.")
    else:
        print("Число нуль.")
```

З наведеного коду програми можна бачити, що вона містить два оператори if-else, один з яких вкладений в інший. Тобто оператор if-else для умови $x < 0$ вкладений (внутрішній) в зовнішній оператор if-else для умови $x > 0$.

Програма буде працювати таким чином:

- Спочатку виконується зовнішній оператор, в якому перевіряється умова $x > 0$.
- Якщо умова $x > 0$ істинна (значення змінної x більше нуля), то виводиться повідомлення, що число додатне.
- Якщо умова $x > 0$ хибна (значення змінної x не більше нуля, тобто нуль або від'ємне), переходимо до виконання вкладеного оператора if- else. Перевіряється умова $x < 0$.
- Якщо умова $x < 0$ істинна (значення змінної x менше нуля), то виводиться повідомлення, що число від'ємне.

- Якщо умова $x < 0$ хибна (значення змінної x рівне нулю, оскільки не виконалась умова $x > 0$, і не виконалась умова $x < 0$), то виводиться повідомлення, що число нуль.

Проте в мові Python передбачений спрощений запис для виконання таких вкладених перевірок. Для їх реалізації використовується оператор `if-elif-else`, яка має вигляд:

```
if Логічний_вираз_1:
    Блок_інструкцій_1
elif Логічний_вираз_2:
    Блок_інструкцій_2
elif Логічний_вираз_3:
    Блок_інструкцій_3
...
[else:
    Блок_інструкцій_N]
```

Оператор працює наступним чином. Обчислюється значення логічного виразу 1. Якщо значення логічного виразу 1 істинне, то виконується блок_інструкцій_1. Якщо ж значення логічного виразу 1 хибне, то відбувається перехід до обчислення значення логічного виразу 2. Якщо значення логічного виразу 2 істинне, то виконується блок_інструкцій_2 і так далі. Блок_інструкцій_N буде виконаний в тому випадку, якщо жодний з логічних виразів не був істинним.

Враховуючи оператор `if-elif-else`, перепишемо нашу програму.

Спосіб 3. Використання оператора `if-elif-else`.

```
x = int(input("Введіть число: "))
if x > 0:
    print("Число додатне.")
elif x < 0:
    print("Число від'ємне.")
else:
    print("Число нуль.")
```

Оператор `if-elif-else` може використовуватися для заміни оператора `switch-case` або `case` в інших мовах програмування.

5.4. Тримісний оператор if/else

Як і в деяких інших мовах програмування, в мові Python передбачений тримісний оператор if/else, який в окремих випадках більш доцільно використовувати замість оператора if-else. Незважаючи на те, що сфера його застосування більш вузька.

Наприклад, нам необхідно в залежності від деякої умови надати одне чи інше значення змінній (наприклад, змінній a присвоюється значення виразів Y або Z в залежності від істинності умови X). З використанням оператора if-else це запишеться так:

```
if X:
    a = Y
else:
    a = Z
```

Використовуючи тримісний оператор if/else, це можна записати так:

```
a = Y if X else Z
```

5.5. Логічні оператори

Іноді є необхідність будувати більш складні логічні вирази, які будуть містити декілька простих логічних тверджень, та між якими необхідно виконати логічні оператори: І, АБО, НІ (and, or, not).

Операндами операторів and, or та not можуть бути як логічні вирази (результат яких має логічний тип), так і вирази, результат яких не є логічного типу (число, рядок, список і т.д.). Тому в загальному можна сказати, що операндами операторів and, or та not є об'єкти.

Логічний оператор not (НЕ)

Логічний оператор not також називають запереченням.

Використання оператора: not X.

Результатом застосування логічного оператора not є значення логічного типу, яке є запереченням операнда.

Якщо операнд істинний (True, будь-яке число не рівне нулю, або не порожній об'єкт), то за оператором not буде повернуто – False. Якщо операнд

хибний (False, нуль, порожній об'єкт або спеціальний об'єкт None), то за оператором not буде повернуто – True.

Логічний оператор and (І)

Логічний оператор and також називають кон'юнкцією або логічним множенням.

Використання оператора: X1 and X2[and X3 ...].

Результатом застосування логічного оператора and є об'єкт.

При обчисленні оператора and операнди обчислюються зліва направо. Як тільки знайдено перший об'єкт, що має хибне значення (інтерпретується як хибне), він вважається результатом обчислення оператора and, і подальше обчислення завершується. Таке раннє завершення обчислення називається обчисленням за короткою схемою. Якщо серед операндів не знайдено об'єкта, що має хибне значення, то повертається крайній правий об'єкт.

```
>>>0 and 3 #повертає перший хибний об'єкт
0
>>>5 and 4 #повертає крайній правий об'єкт
4
```

Якщо операндами оператора and є логічні вирази, то процес обчислення більш спрощений і його можна описати так. Логічний оператор and повертає істину, якщо всі операнди будуть істинними, якщо ж принаймні один з операндів буде хибним, то логічний оператор and поверне хибу.

```
>>> 2>4 and 45>3
False
```

Логічний оператор or (АБО)

Логічний оператор or також називають диз'юнкцією або логічне додавання. Використання оператора: X1 or X2[or X3 ...].

Результатом застосування логічного оператора or є об'єкт.

При обчисленні оператора or операнди обчислюються зліва направо. Як тільки знайдено перший об'єкт, що має істинне значення (інтерпретується як істинне), він вважається результатом обчислення оператора or і подальше обчислення завершується. Таке раннє завершення обчислення називається обчисленням за короткою схемою. Якщо серед операндів не знайдено об'єкта, що має істинне значення, то повертається крайній правий об'єкт.

```
>>>2 or 3 # повертає перший істинний об'єкт
2
>>> None or 0 # повертає крайній правий об'єкт
0
```

Якщо операндами оператора `or` є логічні вирази, то процес обчислення можна описати так. Логічний оператор `or` повертає істину, якщо принаймні один операнд буде істинним, якщо ж всі операнди будуть хибними, то логічний оператор `or` поверне хибу.

Розв'яжемо невелику задачу. Обчислити значення виразу $1/x$ без використання умовного оператора та обробки винятків. Особливістю даної задачі є те, що у випадку, коли x буде рівне 0, виконання ділення $1/x$ призведе до виникнення помилки. Використовуючи особливості логічних операторів, для розв'язання даної задачі можна записати наступний вираз: `x and 1/x`.

```
>>> x=1
>>> x and 1/x
1.0
>>> x=0
>>> x and 1/x
0
```

Логічні вирази можна комбінувати:

```
>>> 1+3 > 7 # пріоритет операції «+» вище, чим в «>»
False
```

Для більшої зрозумілості можуть використовуватися дужки.

```
>>> (1+3) > 7
False
```

Поміркуйте, що буде виведено в результаті виконання виразу, і чому буде отриманим саме такий результат:

```
>>> 1+(3>7)
1
```

В Python можна перевіряти приналежність інтервалу:

```
>>> x=0
>>> -5<x<10 # еквівалентне: x > -5 and x<10
True
```

5.6. Приклади розв'язування задач

Приклад. Написати програму, за якою будуть знайдені корені квадратного рівняння, заданого своїми коефіцієнтами.

```
import math
a=float(input('Введіть коефіцієнти рівняння a,b,c:'))
b=float(input(''))
c=float(input(''))
d=b**2-4*a*c
if d>=0:
    x1=(-b-math.sqrt(d))/(2*a)
    x2=(-b+math.sqrt(d))/(2*a)
    print('x1={:.4}, x2={:.4}'.format(x1,x2))
else:
    print('Дійсних коренів немає')
```

Приклад. Написати програму, за якою буде визначитися, чи є заданий рік високосним. Відповідно з Григоріанським календарем, рік є високосним, якщо його номер кратний 4, але не кратний 100, а також, якщо він кратний 400.

```
year=int(input('Введіть рік: '))
if (year % 4 == 0 and not year % 100 == 0) or
    (year % 400 == 0):
    print('Рік високосний.')
else:
    print('Рік не високосний.')
```

Приклад. Написати програму, за якою буде визначено, чи є дві клітинки, задані користувачем, одного кольору чи ні. Для задання клітинки шахової дошки необхідно вказати два числа від 1 до 8, які будуть визначати номер рядка та номер стовпця.

	1	2	3	4	5	6	7	8	9	10
1		■		■		■		■		■
2	■		■		■		■		■	
3		■		■		■		■		■
4	■		■		■		■		■	
5		■		■		■		■		■
6	■		■		■		■		■	
7		■		■		■		■		■
8	■		■		■		■		■	
9		■		■		■		■		■
10	■		■		■		■		■	

```

x1=int(input('Номер рядка першої клітинки: '))
y1=int(input('Номер стовпця першої клітинки: '))
x2=int(input('Номер рядка другої клітинки: '))
y2=int(input('Номер стовпця другої клітинки: '))
if 1<=x1<=8 and 1<=y1<=8 and 1<=x2<=8 and 1<=y2<=8:
    k1=x1%2==y1%2
    k2=x2%2==y2%2
    if k1==k2:
        print('Клітинки одного кольору.')
    else:
        print('Клітинки різного кольору.')
else:
    print('Невірні вхідні дані.')

```

6. Циклічні оператори

Досить часто і в житті, і при написанні програм існує необхідність повторення деякої дії певної кількості раз. Тобто при написанні програм може знадобитися певна конструкція, за якою можна буде організувати повторне виконання операторів. Таку конструкцію називають конструкцією повторення або циклом. А кожну повторену дію – кроком циклу або ітерацією. Отже, можна зазначити, що цикл у програмуванні – це

повторюване виконання одних і тих самих простих або складених операторів.

Всі цикли складаються з заголовку та тіла циклу. Заголовок циклу відповідає за налагодження циклу, тобто умову повторення циклу. Тіло ж циклу відповідає за самі дії, які мають повторно виконуватися. Так наприклад, уявімо собі першокласника, який дуже любить морозиво. Йому мама видала певну суму грошей на морозиво. Зрозуміло, він побіг його купувати, але згадав, що помножити та ділити він не вміє. Як же йому вирішити цю проблему? Спочатку він перевірить, чи вистачить йому грошей на купівлю пачки морозива. Якщо так, то він її купить і знову погляне на залишок грошей. В цьому прикладі можна виділити заголовок циклу – поки грошей достатньо, і тіло циклу – купівля морозива.

6.1. Цикл з передумовою (Цикл `while`)

Цикл з передумовою є одним з самих універсальних циклів в мові Python, але достатньо повільний. Цикл є циклом з передумовою, оскільки умова записується і перевіряється до тіла циклу. Цикл з передумовою ще називають циклом `While`, оскільки саме з цього ключового слова він починається.

Синтаксис оператора циклу `while`:

```
while Логічний_вираз:  
    Блок_інструкцій
```

Логічний вираз також називають умовою виконання циклу, а блок інструкцій – тілом циклу, яке може містити довільні оператори. За циклом `while` виконується вказаний набір інструкцій до тих пір, поки умова циклу істинна.

При виконанні циклу `while` спочатку обчислюється значення логічного виразу, якщо це значення є істинним (істинність умови визначається так само як і в операторі `if`), то виконується тіло циклу і відбувається повернення до перевірки логічного виразу. Процес продовжується доти, поки значення логічного виразу не стане хибним. Після цього робота циклу завершиться і відбувається перехід до інструкції після тіла циклу `while`.

Якщо при першому обчисленні значення логічного виразу є хибним, тіло циклу не виконується жодного разу.

Щоб цикл закінчив роботу, в його тілі повинен бути оператор, що впливає на значення логічного виразу. Окрім того логічний вираз має бути коректним, тобто його значення повинно бути визначеним ще до першої перевірки.

Як правило цикл `while` використовується, коли неможливо визначити точне значення кількості проходів використання циклу.

6.1.1. Приклади розв'язування задач

Приклад. Обчислити значення $\sum_{n=1}^{\infty} \frac{1}{n^2}$ з точністю до 10^{-6} .

Тут сумування проводиться до тих пір, поки черговий доданок не стане менше заданої точності обчислень.

```
s=0
n=1
x=1
while abs(x)>=1e-6:
    s=s+x
    n=n+1
    x=1/n**2
print(s)
```

Приклад. Визначити кількість цифр натурального числа `n`:

```
n = int(input())
length = 0
while n > 0:
    n = n // 10
    length = length + 1
print(length)
```

Приклад. Обчислити суму непарних додатних чисел, менших за `n`.

```
n=int(input())
s=0
i=1
while i<n:
    s=s+i
```

```
i=i+2
print('Сума непарних чисел =', s)
```

6.2. Тип діапазон (range)

Тип діапазон (range) є незмінюваною послідовністю цілих чисел.

Для задання діапазону призначені функції:

range(stop) – задання послідовності цілих чисел від 0 до stop-1 з кроком 1.

range(start, stop[, step]) - задання послідовності, яка є арифметичною прогресією від start до stop-1 з кроком step. Якщо параметр step опущений, він за замовчуванням дорівнює 1.

За функцією range(5) отримаємо діапазон в з елементів 0, 1, 2, 3, 4

За функцією range(1, 5) отримаємо діапазон в з елементів 1, 2, 3, 4

За функцією range(0, 10, 3) отримаємо діапазон в з елементів 0, 3, 6, 9

Для отримання діапазону, в якому значення будуть зменшуватися, необхідно використовувати функцію range з трьома параметрами. Третій параметр має бути від'ємним, а перший більшим ніж другий.

За функцією range(0, -5, -2) отримаємо діапазон в з елементів 0, -2, -4

Проте вивести на екран елементи утвореного діапазону звичайними методами неможливо, так:

```
>>> r = range(1, 5)
>>> r
range(1, 5)
```

Можна лише перевірити приналежність деякого числа до діапазону, використовуючи оператор in.

```
>>> 4 in r
True
>>> 6 in r
False
```

6.3. Цикл for

Окрім циклу з передумовою, в мові Python є цикл for, за яким надається можливість перебору всіх елементів з деякого набору

(послідовності, бінарної послідовності, рядка, множини, словника, файлу). В загальному можна зазначити, що використовуваним набором в циклі `for`, може будь який набір, що підтримує ітерування. Перебір елементів можна пояснити так. У нас є набір, що складається з ряду елементів. При переборі ми спочатку беремо з даного набору перший елемент, і в тілі циклу виконуємо над ним визначені дії. Потім беремо другий елемент, і над ним знову виконуємо ті ж дії. І так далі продовжуємо над всіма елементами набору. При такому опрацюванні не потрібно турбуватися про індекси елементів і їх кількість.

Іншими словами, можна зазначити, цикл `for` являє собою формальний запис інструкції виду: «Виконати операцію X для всіх елементів, що входять в набір M». Тому цикл `for` інколи називають циклом перегляду.

Синтаксис оператора циклу `for` записується так:

```
for Ідексна_змінна in Послідовність:  
    Блок_інструкцій
```

На початку індексній змінній надається значення першого елемента послідовності, потім виконується тіло циклу (блок інструкцій) і індексній змінній надається значення наступного елемента послідовності. Так продовжується доти, поки індексній змінній послідовно не будуть надані значення всіх елементів послідовності. Тобто індексна змінна буде пробігати всі елементи послідовності.

Як правило, цикли `for` використовуються для виконання операцій над всіма елементами послідовності або виконання операцій таку кількість разів, яка відповідає кількості елементів в послідовності.

Цикл `for` дещо складніший і менш універсальний, але виконується значно швидше циклу `while`.

6.3.1. Приклади розв'язування задач

Приклад. Вивести на екран квадрати додатних цілих чисел, менших за `n`.

```
n=int(input())  
for i in range(1,n):  
    print(i**2)
```

Якщо значення змінної n буде рівне нулю або від'ємне, то тіло циклу не виконається жодного разу.

Приклад. Надрукувати числа від 10 до 1.

```
for i in range(10, 0, -1):  
    print(i)
```

Приклад. Обчислити суму ряду $\sum_{i=1}^n \frac{i}{2i+1}$ для заданого n .

```
n=int(input())  
s=0  
for i in range(1, n+1):  
    s=s+i/(2*i+1);  
print('Сума =', s)
```

Приклад. Обчислити суму непарних додатних чисел з проміжку $[n; m]$.

```
n=int(input())  
if n%2==0:n=n+1  
m=int(input())  
s=0  
for i in range(n, m+1, 2):  
    s=s+i  
print('Сума непарних чисел =', s)
```

В даній програмі спочатку уточнюється початок проміжку таким чином, щоб початком було перше непарне число з заданого проміжку. Потім за функцією `range(n, m+1, 2)` формується діапазон всіх непарних чисел з заданого проміжку і відбувається їх підсумовування.

6.4. Інструкції управління циклами

Оператор `continue`

Оператор *continue* призначений для переривання поточної ітерації циклу і переходу до наступної. Тобто оператори, що будуть іти в тілі циклу після виклику `continue`, на даному кроці виконуватися не будуть.

Приклад. Обчислити суму непарних додатних чисел з проміжку $[n; m]$, не включати до суми чисел, які діляться націло на 5.

```

n=int(input())
if n%2==0:n=n+1
m=int(input())
s=0
for i in range(n,m+1,2):
    if i%5==0:
        continue
    s=s+i
print('Сума непарних чисел =',s)

```

При виконанні тіла циклу, коли значення змінної i буде ділитися націло на 5 (залишок від цілочисельного ділення i на 5 дорівнює 0), відбувається перехід на наступний елемент послідовності.

Оператор `break`

Оператор ***break*** призначений для дострокового припинення роботи циклу (`for` або `while`), тобто зупинки виконання тіла циклу, навіть якщо умова виконання циклу ще не набула значення `False` або послідовність елементів не закінчилась.

Зрозуміло, інструкцію `break` варто викликати тільки всередині інструкції `if`, тобто вона повинна виконуватися тільки при виконанні якоїсь особливої умови.

Приклад. Обчислити суму непарних додатних чисел з проміжку $[n; m]$. При додаванні до суми числа, яке діляться націло на 5, припинити підсумовування.

```

n=int(input())
if n%2==0:n=n+1
m=int(input())
s=0
for i in range(n,m+1,2):
    s=s+i
    if i%5==0:
        break
print('Сума непарних чисел =',s)

```

6.5. Блок `else` в циклах

Блок *else* може використовуватися як додатковий блок циклів `while` та `for`, сфера застосування якого досить схожа до застосування однойменного блоку в конструкціях обробника винятків (`try-except`) та умовного оператора (`if-else`), тобто «якщо цього виконати не можна, то (інакше) виконати це».

Синтаксис операторів циклу з блоком `else`:

```
while Логічний_вираз:
    Блок_інструкцій_1
else:
    Блок_інструкцій_2
Або
for Ідексна_змінна in Послідовність:
    Блок_інструкцій_1
else:
    Блок_інструкцій_2
```

Інструкція всередині блоку `else` (Блок_інструкцій_2) виконується в тому випадку, коли цикл завершився згідно з умовою повторення циклу, вказаною в заголовку циклу (для циклу `while` у випадку, коли умова циклу стала хибною, для циклу `for` – коли були перебрані всі елементи послідовності).

Іншим варіантом завершення циклу є вихід з циклу за виконанням оператора `break` і в такому випадку інструкції блоку `else` не виконуються. Отже використання `else` доцільне тільки разом з інструкцією `break`.

Можливе використання оператора `else` в циклах - реалізація пошукових циклів. Наприклад, необхідно виконати пошук певного елемента, а у випадку його відсутності повідомити користувача про його відсутність.

```
for obj in objects:
    if obj.key == search_key:
        found_obj = obj
        break
else:
    print('No object found.')
```

Приклад. Обчислити суму непарних додатних чисел з проміжку [n; m]. При додаванні до суми числа, яке ділиться націло на 5, припинити підсумовування. Вивести додаткове повідомлення, якщо результуюча сума містить доданок, який ділиться націло на 5.

```
n=int(input())
if n%2==0:n=n+1
m=int(input())
s=0
for i in range(n,m+1,2):
    s=s+i
    if i%5==0:
        break
else:
    print('Сума не містить доданку, що ділиться на
5')
print('Сума непарних чисел =',s)
```

6.6. Вкладені цикли

В усіх операторах циклу мови Python оператор, який є тілом циклу, може сам бути оператором циклу або містити у собі оператор циклу. Утворена конструкція називається вкладеним циклом.

Приклад. За даним натуральним $n \leq 9$ вивести сходи з n сходинок, де i -а сходинка складається з чисел від 1 до i без пропусків.

```
n=int(input('Вкажіть n'))
for i in range(1,n+1):
    for j in range(1,i+1):
        print(j,end=' ')
    print()
```

Приклад. Обчислити суму ряду $\sum_{i=1}^{10} \sum_{j=1}^5 \frac{1}{i+j^2}$.

```
s=0
for i in range(1,11):
    for j in range(1,6):
```

```
s=s+1/(i+j**2);  
print('Сума =', s)
```

7. Структури даних

Для спрощення написання і виконання програми буває зручно окремі дані об'єднувати в певні структури. Від того, наскільки вдало будуть вибрані ці структури, суттєво залежить ефективність програми.

Структури даних - це спосіб організації даних. У мові Python існують вбудовані структури даних, серед яких є: послідовності (списки, кортежі, діапазони), бінарні послідовності, рядки, множини, словники.

Для вбудованих структур даних в мові Python передбачений набір стандартизованих функцій.

len(iterable). Повертає число елементів (довжину) iterable.

max(iterable, *, default=obj, key=func). Повертає максимальний елемент із iterable.

```
>>> max([2,3,4])
```

```
4
```

min(iterable, *, default=obj, key=func). Повертає мінімальний елемент із iterable.

```
>>> min([2,3,4])
```

```
2
```

sum(iterable[, start]). Повертає суму членів числового iterable, починаючи з елемента з індексом start. За замовчуванням start = 0.

```
>>> sum([2,3,4])
```

```
9
```

map(func, *iterables). Застосовує функцію func до кожного елемента із iterable. Результатом є об'єкт, що підтримує ітерування (ітератор).

```
>>> list(map(bin, [1,3,5]))
```

```
['0b1', '0b11', '0b101']
```

enumerate(iterable, start=0). Повертає кортеж (порядковий_номер_елемента, значення_елемента), отриманий з iterable.

```
>>> a=['a','b','c']
>>> for i, v in enumerate(a): print(i, v)
0 a
1 b
2 c
```

7.1. Списки

Список (list) – це структура даних для зберігання елементів (об'єктів) не обов'язково одного типу. Це частково схоже на масиви в інших мовах програмування, але головною особливістю є те, що елементами списку в мові Python можуть бути елементи різних типів. Список є змінюваним типом даних. Списки записуються як перелік елементів, розділених комою та взятих у квадратні дужки: [1, 2, 3, 'Hello'].

7.1.1. Задання списків

Для задання порожнього списку можна скористатися однією з наступних команд:

```
>>> a=[]
>>> a=list()
```

Задання списку з наперед заданим набором елементів:

```
>>> a=[1, 2, 3, 4]
>>> b=['Hello', 2, True]
```

Створення списку з інших структур даних

Список можна отримати з елементів об'єкту, що може ітеруватися (діапазон, рядок, словник, множина, кортеж, файл і т.д.) використавши функцію `list([iterable])`:

```
>>> b=list('Hello')
>>> print(b)
['H', 'e', 'l', 'l', 'o']
>>> c=list(range(10))
>>> print(c)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Об'єднання списків

Нові списки можуть бути створені методом конкатенації (об'єднання) декількох списків в один. Для цього використовується перевизначена операція додавання («+»).

```
>>> b=[1, 3, 5, 7, 9]
>>> d=[10, 11, 12]
>>> c=b+d
>>> print(c)
[1, 3, 5, 7, 9, 10, 11, 12]
```

Багаторазове повторення елементів

Аналогічно до перевизначеної операції («+»), для списків в мові Python перевизначена і операція множення «*». Якщо виконати операцію «*» списку *a* на ціле число *n*, то в результаті буде отриманий список, що складається з *n* повторень списку *a*:

```
>>> a=[0]*5
>>> print(a)
[0, 0, 0, 0, 0]
```

Введення або генерування елементів списку

Числові елементи списку можна згенерувати випадковим чином, скориставшись функціями модуля `random`, наприклад:

```
import random
n=int(input('Введіть кількість елементів списку ='))
a=[]
for i in range(n):
    x=random.randint(1,100)
    a.append(x)
```

Для випадку, коли необхідно створити список з елементів, що будуть задаватися користувачем, можна скористатися наступним фрагментом коду:

```
n=int(input('Введіть кількість елементів списку = '))
a=[]
for i in range(n):
    x=int(input('a[{}]='.format(i)))
    # або x=int(input('a['+str(i)+'']='))
```



```
a.append(x)
```

Якщо ж елементи списку будуть задаватися одним рядком через пропуск, то для формування такого списку можна скористатися наступним фрагментом коду:

```
a=list(map(int, input('Введіть елементи списку через  
пропуск: ').split()))
```

Генератори списків

Для задання списків також можуть бути використані так звані *генераторні списки* (List Comprehensions), які інколи також називають «абстракція списків» або «спискові включення». Генераторні списки є частиною синтаксису мови Python, яка надає простий спосіб побудови списків.

Генератори списків забезпечують лаконічний спосіб створення списку у тому випадку, коли кожен елемент списку є результатом деякої операції, застосованої до кожного елемента іншого списку (послідовності), або створення списку з тих елементів, які задовольняють конкретну умову.

Генератор списку складається з квадратних дужок, що містять вираз формування елемента списку, і циклу for, за яким відбувається перебір елементів іншої «базової» послідовності. Окрім того, елементи, що перебираються циклом for, можуть бути обмежені наявністю умови if.

Створення списку цілих чисел від 0 до 10:

```
>>> a=[i for i in range(11)]  
>>> print(a)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Створення списку, який містить 5 нулів:

```
>>> a=[0 for i in range(5)]  
>>> print(a)  
[0, 0, 0, 0, 0]
```

Створення списку з елементів, які є квадратами чисел від 1 до 4:

```
>>> a=[i ** 2 for i in range(1, 5)]  
>>> print(a)  
[1, 4, 9, 16]
```

Створення списку цілих непарних чисел, менших за 10:

```
>>> a=[i for i in range(10) if i%2==1]
```

```
>>> print(a)
[1, 3, 5, 7, 9]
```

Створення списку з 10 елементів, що заповнений випадковими числами від 1 до 9:

```
>>> import random
>>> a=[random.randrange(1, 10) for i in range(10)]
>>> print(a)
[8, 7, 1, 3, 4, 6, 1, 3, 7, 1]
```

Створення списку, елементами якого будуть пари чисел, які є елементами двох інших списків:

```
>>> a=[ (x, y) for x in [1, 2, 3] for y in [4, 5]]
>>> print(a)
[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
```

Попередня команда еквівалентна, до

```
a=[]
for x in [1,2,3]:
    for y in [4, 5]:
        a.append((x, y))
print(a)
[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
```

7.1.2. Доступ до елементів списку. Зрізи

Доступ до елементів списку відбувається за їх індексами. Як і в багатьох інших мовах, нумерація елементів починається з нуля. Для того, щоб звернутися до елемента списку, необхідно вказати ім'я змінної списку та в квадратних дужках індекс необхідного елемента (ім'я_списку[індекс]).

```
>>> b=[1, 3, 5, 7, 9]
>>> b[1]
3
```

При спробі доступу до неіснуючого індексу виникає виняток `IndexError`.

```
>>> b[5]
Traceback (most recent call last):
```

```
File "<pyshell#27>", line 1, in <module>
    b[5]
```

```
IndexError: list index out of range
```

Індекси можуть бути від'ємними, в такому випадку нумерація буде відбуватися з кінця (кількість елементів списку + від'ємний індекс).

```
>>> b[-1]
```

```
9
```

Якщо ж розглянути всі індекси для списку [1, 3, 5, 7, 9], отримаємо:

список b	1	3	5	7	9
Індекс	b[0]	b[1]	b[2]	b[3]	b[4]
Індекс	b[-5]	b[-4]	b[-3]	b[-2]	b[-1]

Можна перевірити приналежність деякого елемента до списку, використовуючи оператор in (значення in ім'я_списку).

```
>>> a=[1, 3, 5, 7]
```

```
>>> 3 in a
```

```
True
```

```
>>> 4 in a
```

```
False
```

Для протилежної перевірки, що елемент не належить списку, може бути використаний оператор not in.

```
>>> 4 not in a
```

```
True
```

Раніше зазначалося, що використовуючи цикл for, можна перебрати всі елементи послідовності, а отже і списку. Тому команда виведення елементів списку окремими рядками буде виглядати так:

```
>>> for i in b:
    print(i)
```

А команда виведення елементів списку через пропуск, так:

```
>>> for i in b:
    print(i, end=' ')
```

Зрізи (slice)

Досить часто необхідно отримати не один елемент списку за індексом, а деякий набір елементів за певним простим правилом. Наприклад: перші 5 елементів, кожен другий елемент. В таких завданнях замість перебору в циклі набагато зручніше використовувати так званий зріз (slice, slicing). Зріз – отримання з даного списку набору з його елементів. Зріз списку також є списком. Отримання одного елемента списку є найпростішим варіантом зрізу.

Задати зріз можна одним з двох варіантів:

```
item[start: stop].
```

```
item[start: stop: step].
```

Для списку `item` береться зріз від індексу `start`, до `stop` (не включаючи його), з кроком `step`. Також при записі зрізу деякі, а можливо і всі параметри можуть бути опущені (знаки двокрапки в записі все рівно залишаються). У випадку відсутності деяких параметрів їх значення встановлюється за замовчуванням, а саме: `start = 0`, `stop =` кількості елементів списку, `step = 1`.

```
>>> a = [1, 3, 8, 7]
>>> a[1:3]
[3, 8]
```

Якщо опустити другий параметр (залишивши двокрапку), то зріз береться до кінця рядка. Наприклад, щоб отримати зріз без першого елемента, можна записати `a[1:]`. Якщо ж опустити перший параметр, то отримаємо зріз, який містить вказану кількість елементів, що йдуть на початку списку.

```
>>> a[1:]
[3, 8, 7]
>>> a[:3]
[1, 3, 8]
>>> a[:]
[1, 3, 8, 7]
```

При заданні значення третього параметра, рівного 2, у зріз потрапить кожний другий елемент списку.

```
>>> a[::2]
```

```
[1, 8]
>>> a[1::2]
[3, 7]
```

У випадку, якщо параметри `start` і `stop` мають від'ємні значення, то нумерація відбуватиметься з кінця (кількість символів рядка + від'ємний індекс). Наприклад, `a[-2:]` - це список з останніх двох елементів.

```
>>> a[-2:]
[8, 7]
>>> a[:-2]
[1, 3]
>>> a[-3:-1]
[3, 8]
```

Якщо параметр `step` має від'ємне значення, то зріз береться справа наліво.

```
>>> a[::-1]
[7, 8, 3, 1]
>>> a[-2::-1]
[8, 3, 1]
>>> a[1:4:-1]
[]
```

В останньому прикладі був отриманий порожній список, так як `start < stop`, а `step` від'ємний. Те ж саме відбудеться, якщо діапазон індексів виявиться за межами списку.

```
>>> a[10:20]
[]
```

7.1.3. Зміна та вилучення елементів списку

Оскільки списки є змінюваним типом даних, то окремі елементи списку можуть бути змінені чи вилучені.

В цьому пункті буде розглянуто виконання дій над елементами списків без застосування спеціальних методів списків, які подані далі.

Зміна елементів списку

Враховуючи, що список є змінюваним типом даних, то за необхідності можна змінити один чи кілька елементів списку. Для зміни елемента списку необхідно вказати ім'я змінної списку та в квадратних дужках індекс необхідного елемента та виконати присвоєння нового значення (ім'я_списку[індекс]=нове_значення).

```
>>> b=[1, 3, 5, 7, 9]
>>> b[2]=10
>>> print(b)
[1, 3, 10, 7, 9]
```

Змінювати також можна не один елемент, а відразу декілька, використовуючи зрізи.

```
>>> b[1:3]=[11,12]
>>> print(b)
[1, 11, 12, 7, 9]
```

Також використовуючи зріз, елементи можна навіть додавати.

```
>>> b[1:3]=[2,3,4,5,6]
>>> print(b)
[1, 2, 3, 4, 5, 6, 7, 9]
>>> b[len(b):]=[11,12,13]
>>> print(b)
[1, 2, 3, 4, 5, 6, 7, 9, 11, 12, 13]
```

Проте варто зауважити, що при спробі додати список елементів не зрізу, а одному елементу, отримаємо заміну цього елемента на окремий список.

```
>>> b[0]=[0, 1]
>>> print(b)
[[0, 1], 2, 3, 4, 5, 6, 7, 9, 11, 12, 13]
```

Вилучення елементів списку

Для вилучення елемента списку за його індексом можна скористатися командою `del (del ім'я_списку[індекс])`.

```
>>> a=[1, 3, 5, 1, 3]
>>> del a[2]
>>> print(a)
```

```
[1, 3, 1, 3]
```

Також команда `del` може бути використана для вилучення зрізу із списку чи очищення всього списку

```
>>> del a[1:3]
>>> print(a)
[1, 3]
>>> del a[:]
>>> print(a)
[]
```

7.1.4. Змінюваність типу список

Змінна, визначена як список, містить посилання на область в пам'яті, яка в свою чергу містить посилання на елементи (об'єкти) цього списку. На відміну від числових типів даних, список є змінюваним типом даних і тому вміст списку може бути змінений, розширений чи зменшений (Рис. 7.1).

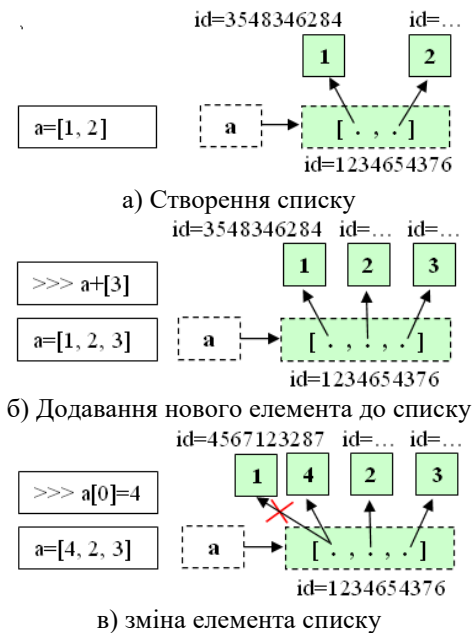


Рис. 7.1. Подання дій з списками

Присвоювання списків

При використанні звичайного присвоювання змінної одного списку іншій змінній відбувається присвоювання новій змінній лише посилання на той же об'єкт в пам'яті, на який послався початковий список. Тобто дві змінні будуть посилатися на один той же об'єкт в пам'яті, і якщо ви будете змінювати один список, то буде змінюватися і інший.

```
>>> a=[1, 3, 5, 7, 9]
>>> b=a
>>> a[1]=2
>>> print(b)
[1, 2, 5, 7, 9]
>>>> id(a)
19733680
>>>> id(b)
19733680
```

Тому для створення нового списку, який буде копією даного, необхідно скористатися зрізом, або функцією `list()`.

```
>>> a=[1, 3, 5, 7, 9]
>>> b=a[:]
>>> a[1]=2
>>> print(b)
[1, 3, 5, 7, 9]
>>> c=list(a)
>>> a[2]=4
>>> print(c)
[1, 2, 5, 7, 9]
>>> id(a)
17285288
>>> id(b)
17343232
>>> id(c)
24285432
```


7.1.5. Методи списків

list.append(x). Додає елемент *x* в кінець списку *list*. Аналогічно командам `a[len(a):]=[x]`.

```
>>> a=[1,2]
>>> a.append(3)
>>> print(a)
[1, 2, 3]
```

list.extend(iterable). Розширює існуючий список *list* за рахунок додавання до нього всіх елементів з *iterable*. Аналогічно командам `a[len(a):]=iterable`.

```
>>> a=[1,2]
>>> a.extend([3,4])
>>> print(a)
[1, 2, 3, 4]
```

list.insert(n, x). Вставляє в список *list* елемент *x* в позицію *n* (індекс елемента, після якого буде вставлений елемент).

```
>>> a=[1,2]
>>> a.insert(1, 5)
>>> print(a)
[1, 5, 2]
>>> a.insert(len(a), 9)
>>> print(a)
[1, 5, 2, 9]
```

list.remove(x). Вилучає перше входження елемента *x* зі списку *list*.

```
>>> a=[1, 2, 3, 1, 2]
>>> a.remove(1)
>>> print(a)
[2, 3, 1, 2]
```

list.pop([n]). Вилучає з списку *list* елемент з позиції *n* та повертає його, як результат виконання функції. Якщо використовувати метод без параметру, то буде вилучений останній елемент списку.

```
>>> a=[1,2,3,4,5]
```

```
>>> print(a.pop(2))
3
>>> print(a.pop())
5
>>> print(a)
[1, 2, 4]
```

list.clear(). Очищує список list (вилучає всі елементи зі списку). Аналогічно до `del a[:]`.

```
>>> a=[1,2,3,4,5]
>>> a.clear()
>>> print(a)
[]
```

list.index(x[, start[, end]]). Повертає індекс першого входження елемента x в зрізі list[start: end] (необов'язкові параметри start та end інтерпретуються як нотації зрізу). Значення, що повертається, є індексом списку list. Якщо елемента в списку не знайдено, виникає виняток ValueError.

```
>>> a=[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print(a.index(3))
2
>>> print(a.index(3,3))
5
```

list.count(x). Повертає кількість входжень елемента x в список.

```
>>> a=[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print(a.count(2))
3
```

list.sort(key=None, reverse=False). Відсортовує елементи списку (аргументи методу можуть бути використані для налаштування сортування). За замовчуванням сортування відбувається за зростанням. Для сортування в зворотному порядку використовуйте параметр `reverse = True`. В результаті сортування змінюється сам список.

```
>>> a=[1, 2, 4, 3, 2, 6, 5]
>>> a.sort()
>>> print(a)
```

```
[1, 2, 2, 3, 4, 5, 6]
>>> a.sort(reverse=True)
>>> print(a)
[6, 5, 4, 3, 2, 2, 1]
```

list.reverse(). Змінює порядок розташування елементів у списку на зворотний. Змінюється сам список.

```
>>> a=[1, 2, 4, 3, 2, 6, 5]
>>> a.reverse()
>>> print(a)
[5, 6, 2, 3, 4, 2, 1]
```

list.copy(). Повертає копію списку. Аналогічно `a[:]`.

```
>>> a=[1, 3, 5]
>>> b=a.copy()
>>> print(b)
[1, 3, 5]
```

7.1.6. Порівняння списків

Списки можна порівнювати між собою. Списки порівнюються поелементно, тому можна порівнювати списки лише в тому випадку, якщо їх відповідні елементи мають однаковий тип чи мають відповідні методи порівняння. При порівнянні використовується лексикографічний порядок: спочатку порівнюються перші два елементи. Якщо вони різні, то вони і визначають результат порівняння; якщо вони рівні, порівнюються наступні два елементи, і т.д., поки одну з двох послідовностей не буде вичерпано. Якщо два порівнюваних елемента самі є списками, то порівняння здійснюється рекурсивно. Якщо всі елементи списків рівні, то списки вважаються рівними. Якщо один список збігається з початком іншого, більш короткий список вважається меншим.

Кілька прикладів порівняння, результатом яких буде істина (True):

```
[1, 2, 3] < [1, 2, 4]
[1, 2, 3, 4] < [1, 2, 4]
[1, 2] < [1, 2, -1]
[1, 2, 3] == [1.0, 2.0, 3.0]
[1, 2, ['aa', 'ab']] < [1, 2, ['abc', 'a'], 4]
```

7.1.7. Вкладені списки

Часто в задачах доводиться зберігати прямокутні таблиці з даними. Такі таблиці називаються матрицями або двовимірними масивами.

В загальному випадку матриця записується так:

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

За аналогією з одновимірними масивами, які в мові Python подаються списками, двовимірний масив можна подати у вигляді списку (двовимірного списку), кожен елемент якого в свою чергу є списком.

```
>>> a=[[1, 2, 3], [4, 5, 6]]
```

Тобто маємо числовий двовимірний масив з двох рядків (перший вимір) та трьох стовпців (другий вимір).

До елементів такого списку можна звертатися так само, як і до елементів звичайного списку, лише враховуючи, що перший вимір (рядки) також є списком.

```
>>> print(a[0])
[1, 2, 3]
>>> print(a[0][1])
2
>>> a[1][0]=10
>>> print(a)
[[1, 2, 3], [10, 5, 6]]
```

За аналогією можна описати і тривимірні списки, і списки з нескінченною вкладеністю.

Враховуючи, що в Python на списки ніяких кількісних обмежень не накладається, то вкладені списки можуть мати різну кількість елементів, тобто `a=[[1, 2], [4, 5, 6]]`.

Для обробки і виведення вкладених списків, як правило, використовують вкладені цикли, кожен з яких буде проходити по відповідному виміру списку. Так, для опрацювання двовимірних списків

необхідно два цикли: перший цикл буде проходити по елементах основного списку, другий цикл буде проходити по елементах вкладених списків.

Задання двовимірних списків

Заповнити двовимірний список (масив) розмірності 4x5 нулями можна декількома способами.

Перший спосіб. Спочатку створити список з 4-х нульових елементів. Далі розширити кожен елемент списку одновимірним списком з 5-и елементів:

```
a=[0] * 4
for i in range(4):
    a[i]=[0] * 5
```

Другий спосіб. Створити порожній список, потім 4 раз додати до нього новий елемент, який є списком з 5-и нулів:

```
a=[]
for i in range(4):
    a.append([0] * 5)
```

Проте, ще простіше скористатися генератором: створити список з 4-х елементів, кожен з яких буде списком з 5-и нулів:

```
a = [[0] * 5 for i in range(4)]
```

Заповнення двовимірного масиву розмірності 4x5 значеннями, заданими користувачем, можна виконати наступним чином:

```
a=[]
for i in range(4):
    b=[]
    for j in range(5):
        x=int(input('a[{} , {}]='.format(i, j)))
        b.append(x)
    a.append(b)
```

Або використовуючи генератори списків:

```
a = [[int(input('a[{} , {}]='.format(i, j))) for j in
range(5)] for i in range(4)]
```

Виведення вкладених списків

Для виведення двовимірного масиву в «природньому» вигляді (елементи першого виміру мають записуватися окремими рядками, а елементи другого виміру в межах одного рядка мають розділятися пропусками) можна скористатися наступним фрагментом програми.

```
for row in a:
    for elem in row:
        print(elem, end=' ')
    print()
```

7.1.8. Приклади розв'язування задач

Приклад. Дано список з 10 випадкових дійсних чисел. Підрахувати суму, добуток і середнє арифметичне елементів списку.

```
import random
v=[random.uniform(1, 100) for i in range(10)]
print('Елементи списку:')
for i in v:
    print(i)
print()
s=sum(v)
d=1
for i in v:
    d=s*i
a=s/len(v)
print('Сума = ',s)
print('Добуток = ',d)
print('Сер. арифм. = ',a)
```

Приклад. Дано списки А, В (кожний з 10 цілих випадкових чисел). Побудувати список С за правилом: $c_i = a_i^2 + b_i^2$.

```
import random
a=[random.randint(1, 100) for i in range(10)]
b=[random.randint(1, 100) for i in range(10)]
print('Елементи списку a: ', a)
print('Елементи списку b: ', b)
```

```

c=[]
for i in range(10):
    c.append(a[i]**2+b[i]**2)
print('Елементи списку c: ', c)

```

Приклад. Дано список з 15 цілих випадкових чисел. Знайти максимальне значення елемента списку та підрахувати кількість елементів списку з таким значенням.

```

import random
a=[random.randint(1, 10) for i in range(15)]
print('Елементи списку a: ', a)
m=max(a)
c=a.count(m)
print('Максимальний елемент списку: ', m)
print('Кількість максимальних елементів: ', c)

```

Приклад. Дано цілочисельну матрицю випадкових цілих чисел розміром $m \times n$ (m і n задаються користувачем). Знайти мінімальний елемент матриці та його індекси.

Розглянемо декілька способів розв'язування даної задачі. Проте всі вони матимуть однаковий початок, в якому буде задаватися розмірність матриці, генеруватиметься матриця і виводитиметься в «природному» вигляді.

```

import random
n=int(input('Введіть кількість рядків матриці: '))
m=int(input('Введіть кількість стовпців матриці: '))
a=[]
for i in range(n):
    b=[]
    for j in range(m):
        x=random.randint(10,99)
        b.append(x)
    a.append(b)
print('Згенерована матриця:')
for row in a:
    for elem in row:

```

```
        print(elem, end=' ')
    print()
```

Спосіб 1. Розв'язування методом перебору.

```
. . .
min_el=a[0][0]
min_i=0
min_j=0
for i in range(n):
    for j in range(m):
        if a[i][j]<min_el:
            min_el=a[i][j]
            min_i=i
            min_j=j
print('Мінімальний елемент: ',min_el)
print('Індекс мінімального елемента
      [{} , {}]'.format(min_i,min_j))
```

Спосіб 2. Розв'язування з використанням методів списків.

```
. . .
min_col=list(map(min, a))
min_el=min(min_col)
min_i=min_col.index(min(min_col))
min_j=a[min_i].index(min_el)
print('Мінімальний елемент: ',min_el)
print('Індекс мінімального елемента
      [{} , {}]'.format(min_i,min_j))
```

Якщо ж матриця буде містити декілька рівних елементів, які і будуть мінімальними, то запропоновані способи не підходять.

Спосіб 3. Пошук індексів для всіх мінімумів

```
. . .
min_col=list(map(min, a))
min_el=min(min_col)
print('Мінімальний елемент: ',min_el)
for i in range(n):
    for j in range(m):
```



```

    if a[i][j]==min_el:
        print('Індекс мінімального елемента
              [{} , {}]'.format(i,j))

```

Приклад. Дано цілочисельну матрицю випадкових цілих чисел розміром 4×4. Написати програму для транспонування матриці (переставлення стовпчиків та рядків).

```

import random
a = [[random.randint(10,99) for j in range(4)] for i
      in range(4)]
print('Згенерована матриця:')
for row in a:
    for elem in row:
        print(elem, end=' ')
    print()
#Тут має міститися блок транспонування матриці
print('Транспонована матриця:')
for row in t:
    for elem in row:
        print(elem, end=' ')
    print()

```

Спосіб 1. Транспонування вкладеними циклами:

```

t = []
for i in range(4):
    t_row = []
    for row in a:
        t_row.append(row[i])
    t.append(t_row)

```

Спосіб 2. Згорання внутрішнього циклу в генератор списків:

```

t = []
for i in range(4):
    t.append([row[i] for row in a])

```

Спосіб 3. Згорання зовнішнього циклу в генератор списків:

```

t=[[row[i] for row in a] for i in range(4)]

```

7.2. Кортежі

Кортеж (tuple) - це незмінна структура даних, яка за своєю будовою дуже схожа на список. Інколи навіть кажуть, що кортеж – це незмінюваний список. Так само, як і список, кортеж може містити елементи різних типів. Кортеж записується, як перелік елементів, розділених комою та взятих в круглі дужки: (1, 3, 5, 'Hello').

Існує кілька причин, коли варто використовувати кортежі замість списків. Першою причиною є можливість захисту даних від випадкової зміни (захист від дурня). Якщо ми отримали набір даних, і є необхідність опрацювати його без зміни даних, то це як раз той випадок, коли доцільно використати кортеж.

Другою причиною є те, що кортежі в пам'яті займають менший об'єм у порівнянні зі списками.

```
>>> lst=[1, 2, 3]
>>> tpl=(1, 2, 3)
>>> print (lst.__sizeof__())
32
>>> print (tpl.__sizeof__())
24
```

Третьою причиною є приріст продуктивності, який пов'язаний з тим, що кортежі працюють швидше, ніж списки (наприклад, операції перебору елементів). Четвертою причиною є можливість використання кортежів в якості ключа у словнику.

```
>>> d = {(1, 1, 1) : 1}
>>> d = {[1, 1, 1] : 1}
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    d = {[1, 1, 1] : 1}
TypeError: unhashable type: 'list'
```

7.2.1. Задання кортежів

Для задання порожнього кортежу можна скористатися однією з наступних команд:

```
>>> a=()
>>> b=tuple()
```

Задання кортежу з наперед заданим набором елементів:

```
>>> a=(1, 3, 5, 'Hello')
```

Окрім того, будь-який набір різних об'єктів, розділених комами та не виділених будь-якими дужками (тобто ні квадратними, ні фігурними, ні круглими) за замовчуванням буде вважатися кортежем:

```
>>> b=1, 3, 5, 'Hello'
>>> print(b)
(1, 3, 5, 'Hello')
```

Проте створення кортежу з одного елемента має певні особливості. Так, ввівши команду `a = (3)`, отримаємо в змінній `a` лише число, а не кортеж

```
>>> a=(3)
>>> print(a)
3
```

Для створення кортежу з одного елемента можна записати:

```
>>> a=(3,)
>>> print(a)
(3,)
```

Створення кортежів з інших структур даних

Кортеж можна отримати з елементів об'єкта, що може ітеруватися (діапазон, рядок, словник, множина, кортеж, файл і т.д.), використавши функцію `tuple([iterable])`:

```
>> b=tuple(range(1,10,2))
>>> print(b)
(1, 3, 5, 7, 9)
>>> b=tuple('Hello')
>>> print(b)
('H', 'e', 'l', 'l', 'o')
```

Об'єднання кортежів

Нові кортежі можуть бути створені методом конкатенації (об'єднання) декількох кортежів. Для цього використовується перевизначена операція додавання («+»), яка використовується як операція конкатенації кортежів.

```
>>> b=(1, 3, 5, 7, 9)
>>> b=b + (10,)
>>> print(b)
(1, 3, 5, 7, 9, 10)
```

Багаторазове повторення елементів

Аналогічно до перевизначеної операції («+») для кортежів, в мові Python перевизначена і операція множення «*». Якщо виконати операцію «*» кортежу *b* на ціле число *n*, то в результаті буде отриманий кортеж, що складається з *n* повторень кортежу *b*:

```
>>> b=(0,)*5
>>> print(b)
(0, 0, 0, 0, 0)
```

7.2.2. Виконання дій над кортежами та їхніми елементами

Доступ до елементів кортежу

Доступ до елементів кортежу здійснюється аналогічно доступу до елементів списку – за їхніми індексами. Тобто для того, щоб звернутися до елемента кортежу, необхідно вказати ім'я змінної кортежу та в квадратних дужках індекс необхідного елемента (ім'я_кортежу[індекс]).

```
>>> a=(1, 3, 5, 7)
>>> print(a[0])
1
```

Те ж саме стосується і зрізів кортежів. Задати зріз можна одним з двох варіантів:

```
tuple[start: stop].
tuple[start: stop: step].
```

```
>>> print(a[1:3])
(3, 5)
```

Можна перевірити приналежність деякого елемента до кортежу, використовуючи оператор `in` (значення `in` ім'я_кортежу).

```
>>> a=(1, 3, 5, 7)
>>> 3 in a
True
```

```
>>> 3 in a
False
```

Але, як вже було сказано - змінювати елементи кортежу не можна. При спробі змінити чи вилучити елемент кортежу виникне виняток `TypeError`.

```
>>> b=(1, 2, 3)
>>> b[1]=15
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    b[1]=15
TypeError: 'tuple' object does not support item
      assignment
>>> del b[1]
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    del b[1]
TypeError: 'tuple' object doesn't support item
deletion
```

Вилучення кортежів

Як було зазначено, вилучити окремий елемент з кортежу неможливо, але можна видалити кортеж повністю, скориставшись командою `del` (`del ім'я_кортежу`).

```
>>> a=(1, 3, 5, 7)
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined
```

Перетворення кортежів у список і назад

На основі існуючого кортежу можна створити список, як і навпаки - зі списку можна створити кортеж. Тобто, використовуючи кортеж для захисту

даних, за необхідності можна перетворити його в список, внести необхідні зміни і знову повернути в кортеж.

```
>>> b=(1,2,3)
>>> b=list(b)
>>> b[1]=5
>>> b=tuple(b)
>>> print(b)
(1, 5, 3)
```

Методи кортежів

Враховуючи незмінюваність кортежів, вони мають лише методи: `tuple.index(x[, start[, end]])` та `tuple.count(x)` призначення яких аналогічне до призначенням однойменних методів списків.

7.3. Словники

Звичайні списки являють собою набір пронумерованих елементів, отже, для звернення до деякого елемента списку необхідно вказати його номер (індекс). Номер елемента в списку однозначно ідентифікує сам елемент. Проте досить часто ідентифікувати дані лише за числовими індексами не завжди зручно.

Словник (dict) – це структура даних, призначена для зберігання довільних об'єктів з доступом за довільним ключем. Дані в словнику зберігаються в форматі ключ=значення. Ключі в межах словника мають бути унікальними, тобто двох однакових ключів в словнику бути не може. Ключ повинен мати незмінюваний тип даних: ціле або дійсне число, рядок, кортеж. Враховуючи зазначену структуру, словник інколи ще називають асоціативним масивом.

Словник записується, як перелік пар ключ : значення, розділених комою та взятих в фігурні дужки: {'A1':2, 'A2':3}.

Словник є змінюваним типом даних: в нього можна додавати нові елементи з довільними ключами і вилучати вже існуючі елементи. При цьому розмір використовуваної пам'яті пропорційний розміру словника. Доступ до елементів словника виконується хоча і повільніше, ніж до звичайного списку, але в цілому достатньо швидко.

7.3.1. Створення словників

Для задання порожнього словника можна скористатися однією з наступних команд:

```
>>> a={}
>>> b=dict()
```

Задання словника з наперед заданим набором елементів:

```
>>> a={'A1':2, 'A2':3}
>>> b=dict(id1=4, id2=8)
>>> print(b)
{'id1': 4, 'id2': 8}
```

Варто відмітити, що у випадку створення словника, в якого ключем має виступати число або кортеж, можливе використання лише першої команди:

```
>>> a={1: 'A1', 2: 'A2'}
>>> a={(1,2,3):2, 'A2':3}
```

Крім того, словник можна створити використовуючи генератори:

```
>>> d={x: x**2 for x in (2, 4, 6)}
>>> print(d)
{2: 4, 4: 16, 6: 36}
```

7.3.2. Виконання дій над елементами словника

В цьому пункті буде розглянуте виконання дій над словниками та їхніми елементами без застосування методів словників, які подані в наступному пункті.

Доступ до значень словника

Враховуючи те, що елементом словника є пара ключ=значення, то, як такого, доступу до елемента словник не має. В словнику передбачена можливість доступу до значення елемента словника за його ключем.

Для того, щоб звернутися до значення елемента словника, необхідно вказати ім'я змінної словника та в квадратних дужках ключ необхідного елемента (ім'я_словника [ключ]).

```
>>> e={'A1':2, 'A2':3}
>>> print(e['A2'])
```

При спробі доступу за неіснуючим у словнику ключем виникає виняток `KeyError`.

```
>>> print(e['A3'])
Traceback (most recent call last):
  File "<pyshell#99>", line 1, in <module>
    print(e['A3'])
KeyError: 'A3'
```

Можна перевірити приналежність деякого ключа до словника, використовуючи оператор `in` (ключ `in` ім'я_словника).

```
>>> e={'A1':2, 'A2':3}
>>> 'A1' in e
True
>>> 'A3' in e
False
```

Також можна перевірити наявність певного значення в словнику, проте така перевірка буде відбуватися за результатом методу `values()`.

```
>>> e={'A1':2, 'A2':3}
>>> 2 in e.values()
True
```

За необхідності можна організувати перебір ключів усіх елементів словника.

```
d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
for k in d:
    print(k, d[k])
```

Зміна значень словника

Враховуючи, що словник є змінюваним типом даних, за необхідності можна змінювати значення його елементів. Для зміни значення елемента словника необхідно вказати ім'я змінної словника та в квадратних дужках ключ та виконати присвоєння нового значення (ім'я_словника[ключ]=нове_значення).

```
>>> e={'A1':2, 'A2':3}
>>> e['A1']=5
>>> print(e)
```



```
{'A1': 5, 'A2': 3}
```

Додавання елементів до словника

Щоб додати елемент до словника, потрібно вказати ім'я змінної словника та в квадратних дужках новий ключ та виконати присвоєння нового значення (ім'я_словника[новий_ключ]=значення).

```
>>> e={'A1':2, 'A2':3}
>>> e['A3']=4
>>> print(e)
{'A1': 2, 'A2': 3, 'A3': 4}
```

Вилучення елемента зі словника

Для видалення елемента зі словника можна скористатися командою `del (del ім'я_словника[ключ])`.

```
>>> e={'A1':2, 'A2':3}
>>> del e['A1']
>>> print(e)
{'A2': 3}
```

7.3.3. Методи словників

`dict.fromkeys(iterable [, value])`. Створює новий словник, ключами якого будуть елементи з `iterable` і однаковим для всіх значенням `value`.

```
>>> d.fromkeys(['a', 'b', 'c'],12)
{'a': 12, 'b': 12, 'c': 12}
```

`dict.update([other])`. Доповнює словник `dict` парами (ключ=значення) з словника `other`, якщо ключ вже присутній в словнику, то його значення оновлюється.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> s={'A1':13, 'A5':15}
>>> d.update(s)
>>> print(d)
{'A1': 13, 'A2': 3, 'A3': 5, 'A4': 7, 'A5': 15}
```

`dict.copy()`. Повертає копію словника `dict`.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> e=d.copy()
>>> print(e)
{'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
```

dict.get(key[, default]). Повертає значення зі словника dict за ключем key. У випадку відсутності елемента з ключем key повертається значення default.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> d.get('A2', 10)
3
>>> d.get('A8', 10)
10
```

dict.setdefault(key[, default]). Повертає значення зі словника dict за ключем key. У випадку відсутності елемента з ключем key повертається значення default, і до словника додається елемент з ключем key і значенням default.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> d.setdefault('A1', 13)
1
>>> d.setdefault('A5', 13)
13
>>> print(d)
{'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7, 'A5': 13}
```

dict.keys(). Повертає ключі елементів словника dict у вигляді об'єкту перегляду словника, що забезпечують динамічний перегляд записів словника.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> d.keys()
dict_keys(['A1', 'A2', 'A3', 'A4'])
```

dict.values(). Повертає значення елементів словника dict у вигляді об'єкту перегляду словника, що забезпечують динамічний перегляд записів словника.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> d.values()
```

```
dict_values([1, 3, 5, 7])
```

dict.items(). Повертає ключі та значення елементів словника `dict` у вигляді об'єкту перегляду словника, що забезпечують динамічний перегляд записів словника. Елементи словника подаються в вигляді кортежів (ключ, значення).

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5}
>>> d.items()
dict_items([('A1', 1), ('A2', 3), ('A3', 5)])
```

dict.pop(key[, default]). Вилучає зі словника `dict` елемент з ключем `key` та повертає його значення, як результат виконання функції. У випадку відсутності елемента з ключем `key` повертається значення `default`. Якщо `default` не вказаний, і елемент з ключем `key` відсутній, то виникає виняток `KeyError`.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> d.pop('A2')
3
>>> print(d)
{'A1': 1, 'A3': 5, 'A4': 7}
```

dict.popitem(). Вилучає і повертає пару (ключ, значення) зі словника `dict`, як результат виконання функції. Пари повертаються в порядку LIFO (last-in first-out). Якщо словник порожній, то виникає виняток `KeyError`.

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> print(d)
{'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> d.popitem()
('A4', 7)
>>> print(d)
{'A1': 1, 'A2': 3, 'A3': 5}
```

dict.clear(). Очищує словник `dict` (вилучає всі елементи зі словнику).

```
>>> d={'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
>>> print(d)
{'A1': 1, 'A2': 3, 'A3': 5, 'A4': 7}
```

```
>>> d.clear()
>>> print(d)
{}
```

7.3.4. Приклади розв'язування задач

Приклад. Написати програму для знаходження відстані між двома точками заданими своїми координатами. Для збереження даних точки скористатися словником.

```
p1={'x':0, 'y':0}
p2={'x':0, 'y':0}
print('введіть координати 1-ї точки: ')
p1['x']=float(input('x:'))
p1['y']=float(input('y:'))
print('введіть координати 2-ї точки: ')
p2['x']=float(input('x:'))
p2['y']=float(input('y:'))
import math
import math
lenP=math.sqrt((p1['x']-p2['x'])**2+(p1['y']-
p2['y'])**2)
print('відстань між точками ({}; {}) та ({}; {}) =
{: .3}'.format(p1['x'], p1['y'], p2['x'],
p2['y'], lenP))
```

7.4. Рядкові величини

З точки зору комп'ютера, текст – це набір символів, кожен з яких має свій код. Аналогічно і в мовах програмування, найменшою текстовою одиницею є символ. Символ – це або буква, або цифра, або інший спеціальний символ (символ пропуску, символ новго рядка і т.д.). А уже весь текст подається у вигляді рядка, який є послідовністю символів. Ми вже зустрічалися з рядками, а саме з рядковими літералами.

Як і більшість мов програмування, мова Python часто використовується для опрацювання текстів: пошуку в тексті, заміни окремих

частин тексту і т.д. Для роботи з текстом у мові Python передбачений спеціальний рядковий тип даних `str`.

Рядковий тип або просто рядки в Python – це послідовність символів, яка використовується для зберігання і подання текстових даних. Користуючись рядковим типом, можна працювати з усім, що може бути подано в текстовому вигляді.

У мові Python символами рядка можуть бути будь-які символи, що підтримуються стандартом Unicode. У Python 3 немає ASCII-рядків, якщо необхідно отримати рядок строго в кодуванні ASCII, необхідно скористатися відповідним методом перекодування.

Варто відмітити, що в мові Python немає типу для символа рядка. Кожен символ рядка є також рядком. Також рядковий тип є незмінюваний.

7.4.1. Рядкові літерали та їх задання

В мові Python існує декілька способів задання рядкових літералів. Наприклад, можна розмістити текст в одинарних або подвійних лапках, і такий запис буде сприйматися, як текстовий літерал:

```
>>> 'Привіт'
'Привіт'
>>> "Привіт"
'Привіт'
```

Наявність двох таких варіантів забезпечує створення рядків, які будуть містити одинарні чи подвійні лапки всередині тексту і однозначно визначати кінці рядка.

```
>>> "Ім'я"
"Ім'я"
>>> 'Національний університет "Чернігівський
    колегіум".'
```

```
'Національний університет "Чернігівський колегіум".'
```

Іншим методом додавання в рядок одинарних чи подвійних лапок є так зване екранування символу. Для екранування символу перед ним записується символ оберненої косої риски (зворотній слеш, бекслеш, `backslash`) `«\»`.

```
>>> 'Ім\ 'я'
"Ім'я"
>>> "Національний університет \"Чернігівський
      колегіум\"."
'Національний університет "Чернігівський колегіум".'
```

Рядкові літерали можна присвоювати змінним, які матимуть рядковий тип, і в подальшому використовувати їх:

```
>>> s='Привіт світ!'
>>> print(s)
Привіт світ!
```

Багаторядкові текстові блоки (Багаторядкові літерали)

При роботі з текстовими даними інколи виникає необхідність в заданні багаторядкового текстового блоку. Такий текстовий блок може використовуватися як багаторядковий коментар до програми чи анотація (синтаксис, на якому будуються підказки в мові Python) до функції.

Для того, щоб деякий текст, записаний в декілька рядків, вважався єдиним текстовим літералом, його необхідно взяти в три одинарні чи три подвійні лапки (''' або """). Окрім того в середині такого рядка можна використовувати одинарні чи подвійні лапки (головне, щоб не було трьох лапок підряд).

```
>>> '''Це дуже довгий
      текст, записаний
      в декілька рядків'''
'Це дуже довгий \nтекст, записаний \nв декілька
      рядків'
>>> print(c)
Це дуже довгий
текст, записинай
в декілька рядків
```

7.4.2. Задання рядків

Для задання порожнього рядку можна скористатися однією з наступних команд:

```
>>> s=' '  
>>> s=str()
```

Використання рядкових літералів

Як зазначалося раніше, рядки можна створювати, використовуючи рядкові літерали:

```
>>> s1='Привіт'  
>>> s2="Привіт"  
>>> c='''Це дуже довгий  
текст, записаний  
в декілька рядків'''
```

Введення рядків

Аналогічно до введення числових даних, введення рядкових даних відбувається з використанням функції `input()`.

```
>>> s=input('Введіть рядок: ')  
Введіть рядок: Привіт  
>>> print(s)  
Привіт
```

Приведення до рядкового типу

Будь який стандартний об'єкт мови Python можна привести до рядкового типу, який йому відповідає, або отримати для нього «неформальне» рядкове подання. Для цього необхідно скористатися функцією `str(object='')`, передавши в якості параметра об'єкт, значення якого необхідно привести до рядкового типу.

```
>>> s=str(5)  
>>> print(s)  
'5'  
>>> s=str(True)  
>>> print(s)  
'True'
```

Об'єднання рядків

Нові рядки можуть бути створені методом конкатенації (об'єднання) декількох рядків. Для цього використовується перевизначена операція додавання («+»), яка використовується як операція конкатенації рядків.

```
>>> s1='Привіт '
>>> s2='світ!'
>>> s=s1+s2
>>> s
'Привіт світ!'
```

Незважаючи на незмінюваність рядкового типу, рядок може бути розширеним, тобто до рядка можна додати деякий інший, використовуючи операцію конкатенації.

```
>>> s='Привіт'
>>> id(s)
10388768
>>> s=s+' Світ!'
>>> print(s)
Привіт Світ!
>>> id(s)
10388768
```

Враховуючи строгу типізацію мови Python і неможливість проводити операції у виразах з даними різних несумісних типів (рядок та число є несумісними типами), виконати об'єднання змінних рядкового та числового типів без додаткових перетворень не є можливим. Проте до проведення такого об'єднання можна привести числове значення до рядкового, а вже потім провести операцію конкатенації для двох рядкових значень.

```
>>> s1='Вавілон '
>>> n=15
>>> s2=str(n)
>>> print(s1+s2)
'Вавілон 15'
```

Багаторазове повторення рядка або дублювання рядка

Аналогічно до перевизначеної операції («+»), в мові Python перевизначена і операція множення «*». Якщо виконати операцію «*» рядка

s на ціле число n, то в результаті буде отриманий рядок, що складається з n повторень рядка s:

```
>>> s="СПАМ" * 5
>>> print(s)
'СПАМСПАМСПАМСПАМСПАМ'
```

7.4.3. Доступ до символів рядку. Зрізи

Кожен символ рядка має свій індекс (порядковий номер). Доступ до символів рядка відбувається за їх індексами. Як і в багатьох інших мовах програмування, нумерація символів рядка починається з нуля. Для того, щоб звернутися до певного символу рядка, необхідно вказати ім'я рядкової змінної та в квадратних дужках індекс необхідного символу (рядок[індекс]).

```
>>> s='Привіт!'
>>> s[0]
'П'
```

Індекси можуть бути від'ємними, в такому випадку нумерація буде відбуватися з кінця (кількість символів рядка + від'ємний індекс).

```
>>> s[-1]
'!'
```

Якщо ж розглянути всі індекси для рядка 'Привіт!', отримаємо:

Рядок s	П	р	и	в	і	т	!
Індекс	s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]
Індекс	s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]

Якщо ж вказати індекс, який виходить за межі рядка (наприклад, s[8] або s[-9]), то генерується виняток: `IndexError: string index out of range`.

Можна перевірити приналежність деякого символу (підрядка) до рядка, використовуючи оператор `in` (підрядок `in` рядком).

```
>>> a=[1, 3, 5, 7]
>>> 3 in a
True
```

```
>>> 3 in a
False
```

Зрізи (slice)

Досить часто, необхідно отримати не один якийсь символ (за індексом), а деякий набір символів за певними простими правилами. Наприклад: перші 5 символів, останні 3 символи, кожен другий символ. В таких завданнях замість перебору в циклі набагато зручніше використовувати так званий зріз (slice, slicing). Зріз – отримання з даного рядка набору з його символів. Зріз рядка також є рядком. Отримання одного символу рядка є найпростішим варіантом зрізу.

Слід пам'ятати, що беручи символ за індексом або зрізом, ми ніяк не змінюємо початковий рядок, ми просто скопіювали його частину для подальшого використання.

Задати зріз можна одним з двох варіантів:

```
str[start: stop].  
str[start: stop: step].
```

Для рядку `str` береться зріз від символу з індексом `start`, до символу з індексом `stop` (не включаючи його), з кроком `step` (тобто будуть взяті символи з індексами `start`, `start + step`, `start + 2 * step` і т. д.). Також при записі зрізу деякі, а можливо і всі параметри можуть бути опущені (знаки двокрапки в записі все рівно залишаються). У випадку відсутності деяких параметрів їх значення встановлюється за замовчуванням, а саме: `start = 0`, `stop =` кількості символів рядку, `step = 1`.

```
>>> S='Привіт!'  
>>> S[2:5]  
'иві'
```

Якщо опустити другий параметр (залишивши двокрапку), то зріз береться до кінця рядка. Наприклад, щоб отримати зріз без перших двох символів, можна записати `S[2:]`. Якщо опустити перший параметр, то отримаємо зріз, який містить вказану кількість символів, що йдуть від початку рядка.

```
>>> S[2:]  
'ивіт!'
```

```
>>> S[:3]
'При'
```

При заданні значення третього параметра, рівному 2, в зріз потрапить кожний другий символ рядку.

```
>>> S[::2]
'Пиі!'
>>> S[1::2]
'рвт'
```

Можна бачити, що взяття зрізу схоже на створення діапазону (`range()`).

Якщо значення параметру `stop` буде перевищувати кількість символів в рядку, воно буде проігнороване.

```
>>> S[2:50]
'ивіт!'
```

У випадку, якщо параметри `start` і `stop` мають від'ємні значення, то нумерація відбувається з кінця (кількість символів рядка + від'ємний індекс). Наприклад, `S[1: -1]` - це рядок без першого і останнього символу (зріз починається символом з індексом 1 і закінчується символом з індексом -1, не включаючи його).

```
>>> S[1:-1]
'ривіт'
>>> S[: -4]
'При'
>>> S[-4:]
'віт!'
```

Якщо параметр `step` має від'ємне значення, то зріз береться зправа наліво.

```
>>> S[::-1]
'!тівірП'
>>> S[-2::-1]
'тівірП'
>>> S[1:4:-1]
''
```

В останньому прикладі був отриманий порожній рядок, так як `start < stop`, а `step` від'ємний.

7.4.4. Виконання дій над рядками та їхніми елементами

Визначення довжини рядка

Як зазначалося раніше, рядки складаються з окремих символів. Для того щоб дізнатися кількість символів у рядку (довжину рядка), необхідно скористатися функцією `len` (рядок).

```
>>> len('Привіт!')
7
```

Зміна елементів рядка

Як було зазначено раніше, рядковий тип є незмінюваним, тобто зміна значення символу рядка чи його вилучення є неможливими.

```
>>> s='Привіт!'
>>> s[0]='п'
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    s[0]='п'
TypeError: 'str' object does not support item
assignment
```

За необхідності можна записати оператори для зміни чи вилучення окремих символів рядка, проте в результаті ми отримаємо новий рядок.

```
>>> s='Hello'
>>> s=s[:1]+'a'+s[2:]
>>> print(s)
Hallo
>>> s=s[:1]+s[3:]
>>> print(s)
Hlo
```

Отримання коду символу та символу за кодом

Як було зазначено раніше, кожен символ текстового рядка має свій код. Інколи виникає необхідність отримання цього коду для певного символу,

або зворотна операція – отримання символу за його кодом. При виконанні таких дій варто пам'ятати, що символи в мові Python є символами Unicode.

Для отримання коду символу можна скористатися функцією `ord(c)`, за якою повертається ціле число, яке відповідає коду цього символу.

```
>>> ord('a')
97
>>> ord('€')
8364
>>> ord('½')
189
```

Зворотною до функції `ord()` є функція `chr(n)`, за якою для цілого числа `n` (від 0 до `1114111=0x10FFFF`) повертається символ (рядок, що є одним символом), для якого `n` є його кодом.

```
>>> chr(97)
'a'
>>> chr(8364)
'€'
>>> chr(189)
'½'
```

7.4.5. escape-послідовності

В попередньому пункті при виведенні рядкової змінної (без використання функції `Print`), яка містила багаторядковий текст, в місцях, де мали бути переходи на новий рядок, можна було бачити пару символів «`\n`». Ця пара символів починається з зворотного слеша і називається escape-послідовністю або керувальною послідовністю. Така послідовність в мові Python не єдина.

Використовуючи escape-послідовності, в рядки можна вставляти символи, які важко ввести з клавіатури. Серед найбільш вживаних escape-послідовностей можна виокремити:

- `\f` – перехід на нову сторінку;
- `\n` – перехід на новий рядок;
- `\r` – переведення каретки;

```
\t – горизонтальна табуляція;  
\v – вертикальна табуляція;  
\0 – символ Null (не є ознакою кінця рядка);  
>>> print('Це речення \nв два рядки')  
Це речення  
в два рядки
```

Отже, за функцією `print()` було розпізнано `escape`-послідовність і виконано перехід на новий рядок.

Проте інколи є необхідність, щоб символ зворотнього слеша не сприймався, як початок `escape`-послідовності. Наприклад, це є важливим при заданні шляху до файлу в операційній системі Windows. Для такого відключення аналізу `escape`-послідовностей необхідно перед відкриваючою лапкою вставити символ `r` (у будь-якому регістрі),

```
>>> S = r'C:\newt.txt'  
>>> print(S)  
C:\newt.txt
```

Проте рядки з відключеним аналізом `escape`-послідовностей не можуть закінчуватися символом зворотнього слеша. Але це можна обійти:

```
>>> S = r'\n\n' + '\\'  
>>> print(S)  
\n\n\  
>>> S = r'\n\n\\'[:-1]  
>>> print(S)  
\n\n\  

```

7.4.6. Методи рядків

Для роботи з рядками в Python передбачена велика кількість вбудованих методів.

При роботі з методами рядків необхідно пам'ятати, що рядковий тип в Python є незмінюваним типом, тому всі рядкові методи повертають новий рядок, не змінюючи даного. Для того, щоб змінити сам рядок, необхідно викликати для нього потрібний метод і присвоїти його результат (рядкове значення) тій же змінній, в якій зберігається поточний рядок: `str=str.метод()`.

str.find(substr [, start [,end]]). Повертає найменший індекс, за яким знаходиться початок підрядка `substr` в зрізі `str[start:end]` (необов'язкові параметри `start` та `end` інтерпретуються як нотації зрізу). Тобто знаходиться перше входження підрядка в рядку. Значення, що повертається, є індексом рядка `str`. Якщо підрядок не знайдено, то повертається значення `-1`.

```
>>> 'habrahabr'.find('r')           # 3
>>> 'habrahabr'.find('r',4)         # 8
>>> 'habrahabr'.find('abr')         # 1
>>> 'habrahabr'.find('Abr')         # -1
```

str.rfind(substr [, start [,end]]). Повертає найбільший індекс, за яким знаходиться початок підрядка `substr` в зрізі `str[start:end]` (необов'язкові параметри `start` та `end` інтерпретуються як нотації зрізу). Тобто знаходиться останнє входження підрядка в рядку. Значення, що повертається, є індексом рядка `str`. Якщо підрядок не знайдено, то повертається значення `-1`.

```
>>> 'habrahabr'.rfind('abr')        # 6
>>> 'habrahabr'.rfind('abr',7)      # -1
```

str.index(substr [, start [,end]]). Повертає найменший індекс, за яким знаходиться початок підрядка `substr` в зрізі `str[start:end]` (необов'язкові параметри `start` та `end` інтерпретуються як нотації зрізу). Тобто знаходиться перше входження підрядка в рядку. Значення, що повертається, є індексом рядка `str`. Якщо підрядок не знайдено, то виникає виняток `ValueError`.

str.rindex(substr [, start [,end]]). Повертає найбільший індекс, за яким знаходиться початок підрядка `substr` в зрізі `str[start:end]` (необов'язкові параметри `start` та `end` інтерпретуються як нотації зрізу). Тобто знаходиться останнє входження підрядка в рядку. Значення, що повертається, є індексом рядка `str`. Якщо підрядок не знайдено, то виникає виняток `ValueError`.

str.startswith(prefix[, start[, end]]). Повертає `True`, якщо зріз `str[start:end]` (необов'язкові параметри `start` та `end` інтерпретуються як нотації зрізу) починається з префіксу `prefix` (`prefix`

може бути кортежем, в такому випадку як префікс перевіряються всі елементи кортежу), інакше – False.

```
>>> 'habrahabr'.startswith('ha')           # True
>>> 'habrahabr'.startswith('ha',1)         # False
>>> 'habrahabr'.startswith('ab',1)         # True
>>> 'habrahabr'.startswith(('ha','bla'))   # True
>>> 'blablacar'.startswith(('ha','bla'))   # True
```

str.endswith(suffix[, start[, end]]). Повертає True, якщо зріз str[start:end] (необов'язкові параметри start та end інтерпретуються як нотації зрізу) закінчується на суфікс suffix (suffix може бути кортежем, в такому випадку як суфікс перевіряються всі елементи кортежу), інакше – False.

```
>>> 'habrahabr'.endswith('abr')            # True
>>> 'habrahabr'.endswith('abr',1,4)        # True
>>> 'habrahabr'.endswith('abr',1,8)        # False
>>> 'habrahabr'.endswith(('abr','car'))    # True
>>> 'blablacar'.endswith(('abr','car'))    # True
```

str.count(substr [, start [,end]]). Повертає кількість входжень підрядка sub в зріз str[start:end] (необов'язкові параметри start та end інтерпретуються як нотації зрізу) без самоперетинів.

```
>>> 'habrahabr'.count('ha')                # 2
>>> 'hahahahahah'.count('hah')            # 3
>>> 'hahahahahah'.count('ha')             # 5
>>> 'hahahahahah'.count('ha',1,6)         # 2
```

str.isalpha(). Повертає True, якщо рядок є непорожнім і складається лише з алфавітних символів, інакше – False. Алфавітні символи – це символи, які належать до категорії Юнікоду "Letter" ("Lm", "Lt", "Lu", "Ll", "Lo").

```
>>> 'Habr'.isalpha()                       # True
>>> ''.isalpha()                           # False
>>> 'Habr1'.isalpha()                      # False
>>> 'Habr%'.isalpha()                      # False
```


str.isdecimal(). Повертає True, якщо рядок є непорожнім і складається лише з десяткових цифр (десяткових символів), інакше – False. Десяткові символи – це символи, які належать до категорії Юнікоду "Number, Decimal Digit" ("Nd").

```
>>> '123'.isdecimal()      # True
>>> ''.isdecimal()         # False
>>> '1.1'.isdecimal()      # False
```

str.isdigit(). Повертає True, якщо рядок є непорожнім і складається лише з цифр, інакше – False. Цифри – це символи, які належать до категорій Юнікоду "Number, Decimal Digit" та "Number, Other" ("Nd", "No"). Наприклад, до них належать цифри надрядкового знаку.

```
>>> '²'.isdigit()          # True
```

str.isnumeric(). Повертає True, якщо рядок є непорожнім і складається лише з числових символів, інакше – False. Числові символи – це символи, які належать до категорії Юнікоду "Number" ("Nd", "No", "NI").

```
>>> 'VII'.isnumeric()      # True
```

str.isalnum(). Повертає True, якщо рядок є непорожнім і складається лише з літеро-числових символів, інакше – False. Літеро-числові символи – це символи, які належать до категорій Юнікоду "Letter" та "Number" ("Lm", "Lt", "Lu", "Ll", "Lo", "Nd", "No", "NI").

```
>>> 'Habr1'.isalnum()      # True
```

str.islower(). Повертає True, якщо рядок містить принаймні одну літеру, і всі літери записані в нижньому регістрі, інакше – False.

```
>>> 'habra habr'.islower() # True
>>> 'habra habr 1'.islower() # True
>>> 'habra Habr'.islower() # False
>>> '1'.islower()          # False
```

str.isupper(). Повертає True, якщо рядок містить принаймні одну літеру, і всі літери записані в верхньому регістрі, інакше – False.

```
>>> 'HABRAHABR'.isupper() # True
>>> 'HABRAHABR 1'.isupper() # True
>>> 'HaBRAHABR 1'.isupper() # False
>>> '1'.isupper()          # False
```

str.istitle(). Повертає True, якщо рядок містить принаймні одну літеру, і літери, записані в верхньому регістрі, не йдуть безпосередньо після літер в нижньому чи верхньому регістрі, а перед групою літер в нижньому регістрі завжди стоїть літера в верхньому регістрі, інакше – False.

```
>>> 'Habra Habr'.istitle()           # True
>>> 'lHabra %Habr'.istitle()         # True
>>> 'Habra habr'.istitle()           # False
>>> 'hAbra Habr'.istitle()           # False
```

str.isspace().

Повертає True, якщо рядок є непорожнім і складається лише з символів пропуску (whitespace), інакше – False. Whitespace символи – це символи, які в базі даних Unicode визначені як «Other» або «Separator». Наприклад, до таких символів належать: пробіл, перехід на нову сторінку (\f), перехід на новий рядок (\n), переведення каретки (\r), горизонтальна та вертикальна табуляції (\t та \v).

```
' '.isspace()                         # True
'\n\t'.isspace()                       # True
''.isspace()                            # False
'Habr!'.isspace()                      # False
```

str.isprintable().

Повертає True, якщо рядок містить лише символи, що друкуються (можуть бути виведені при друці), інакше – False. До символів, що друкуються, не входять символи пропуску (whitespace) окрім пробілу.

```
' '.isprintable()                      # True
'Habra habr!'.isprintable()            # True
'\n\t'.isprintable()                   # False
'Habra\nhabr'.isprintable()            # False
```

str.upper(). Повертає копію рядка, в якому всі літери, записані в нижньому регістрі, будуть приведені до верхнього регістру.

```
>>> 'Habra Habr 5%2=1'.upper()
'HABRA HABR 5%2=1'
```

str.lower(). Повертає копію рядка, в якому всі літери, записані в верхньому регістрі, будуть приведені до нижнього регістру.

```
>>> 'Habra Habr 5%2=1'.lower()
'habra habr 5%2=1'
```

str.swapcase(). Повертає копію рядка, в якому всі літери, записані в верхньому регістрі, будуть приведені до нижнього регістру, а нижньому – до верхнього.

```
>>> 'Habra Habr 5%2=1'.swapcase()
'hABRA hABR 5%2=1'
```

str.title(). Повертає копію рядка, в якому перша літера кожного слова буде приведена до верхнього регістру, а всі інші – до нижнього. Першою літерою слова вважається літера, перед якою не міститься інші літери.

```
>>> 'hAbra Habr 5%2=1'.title()
'Habra Habr 5%2=1'
```

str.capitalize(). Повертає копію рядка, в якому перший символ, якщо він є літерою, буде приведений до верхнього регістру, а всі інші літери до нижнього.

```
>>> 'hAbra Habr 5%2=1'.capitalize()
'Habra habr 5%2=1'
>>> '5%2=1 hAbra Habr'.capitalize()
'5%2=1 habra habr'
```

str.replace(old, new[, count]). Повертає копію рядка, в якому всі входження підрядка `old` будуть замінені на новий підрядок `new`. Якщо задано параметри `count`, то буде виконано не більше чим `count` замін.

```
>>> 'habrahabr'.replace('a', 'A')
'hAbrAhAbr'
>>> 'habrahabr'.replace('a', 'A', 2)
'hAbrAhabr'
```

str.lstrip([chars]). Повертає копію рядка з вилученими початковими символами, вказаними в рядку `chars`. Якщо параметр `chars` відсутній або `None`, то вилучаються пропуски.

```
>>> ' habrahabr '.lstrip()
'habrahabr '
>>> 'www.habr.com'.lstrip('cmowa.')
'habr.com'
```

str.rstrip([chars]). Повертає копію рядка з вилученими кінцевими символами, вказаними в рядку `chars`. Якщо параметр `chars` відсутній або `None`, то вилучаються пропуски.

```
>>> ' habrahabr '.rstrip()
' habrahabr '
>>> 'www.habr.com'.rstrip('cmowa.')
'www.habr'
```

str.strip([chars]). Повертає копію рядка з вилученими початковими та кінцевими символами, вказаними в рядку `chars`. Якщо параметр `chars` відсутній або `None`, то вилучаються пропуски.

```
>>> ' habrahabr '.strip()
'habrahabr'
>>> 'www.habr.com'.strip('cmowa.')
'habr'
```

str.expandtabs(tabsize=8). Повертає копію рядка, в якому всі символи табуляції (`\t`) замінюються декількома пропусками, в залежності від стовпця табулювання і заданого розміру табуляції `tabsize`.

```
>>> '\t1\t10\t100'.expandtabs()
'      1      10      100'
>>> '\t1234567\t10'.expandtabs()
'      1234567 10'
>>> '\t12345678\t10'.expandtabs()
'      12345678      10'
>>> '\t1\t10\t100'.expandtabs(4)
'  1  10  100'
```

str.split(sep=None, maxsplit=-1). Повертає список слів, які отримуються розбиттям рядка за роздільником рядком `sep`. Якщо параметр `sep=None`, то роздільником буде виступати пропуск, і результуючий список не буде містити порожніх елементів. Якщо задано параметр `maxsplit`, то буде виконано не більше чим `maxsplit` розбиттів (результуючий список буде мати не більше `maxsplit+1` елемент).

```
>>> 'habrahabr'.split('a')
['h', 'br', 'h', 'br']
```

```
>>> 'haabrahabr'.split('a')
['h', '', 'br', 'h', 'br']
>>> 'habrahabr'.split('ab')
['h', 'rah', 'r']
>>> 'habrahabr'.split('a',2)
['h', 'br', 'habr']
```

str.join(iterable). Повертає рядок, який є результатом конкатенації всіх рядків з `iterable`. Під час конкатенації між рядковими елементами `iterable` буде розміщений рядок `str`. Якщо `iterable` містить принаймні одне нерядкове значення, то генерується виняток `TypeError`.

```
>>> '..'.join(['1', '2'])
'1..2'
>>> '..'.join('hello')
'h..e..l..l..o'
```

str.partition(sep). Повертає кортеж з трьома рядковими значеннями, які є частинами рядку `str`. Першим елементом є частина рядка `str`, що міститься до першого входження роздільника `sep`. Другим елементом є сам роздільник `sep`. Третім елементом є частина рядка `str`, що міститься після роздільника `sep`. Якщо роздільника `sep` в рядку `str` не знайдено, то першим елементом кортежу буде сам рядок `str`, а другий та третій елементи будуть порожніми рядками.

```
>>> 'habrahabr'.partition('ra')
('hab', 'ra', 'habr')
>>> 'habrahabr'.partition('a')
('h', 'a', 'brahabr')
>>> 'habrahabr'.partition('car')
('habrahabr', '', '')
```

str.rpartition(sep). Повертає кортеж з трьома рядковими значеннями, які є частинами рядку `str`. Першим елементом є частина рядка `str`, що міститься до останнього входження роздільника `sep`. Другим елементом є сам роздільник `sep`. Третім елементом є частина рядка `str`, що міститься після роздільника `sep`. Якщо роздільника `sep` в рядку `str` не

знайдено, то першим елементом кортежу буде сам рядок `str`, а другий та третій елементи будуть порожніми рядками.

```
>>> 'habrahabr'.rpartition('a')
('habrah', 'a', 'br')
```

`str.ljust(width, fillchar=" ")`. Повертає копію рядка `str`, доповненого справа символами `fillchar` до довжини `width`. Якщо довжина рядка більша або рівна `width`, повертає оригінальний рядок.

```
>>> 'habr'.ljust(7)
'habr   '
>>> 'habr'.ljust(7, '_')
'habr___'
```

`str.rjust(width, fillchar=" ")`. Повертає копію рядка `str`, доповненого зліва символами `fillchar` до довжини `width`. Якщо довжина рядка більша або рівна `width`, повертає оригінальний рядок.

```
>>> 'habr'.rjust(7)
'   habr'
>>> 'habr'.rjust(7, '_')
'___habr'
```

`str.center(width[, fillchar])`. Повертає копію рядка `str`, доповненого зліва та справа символами `fillchar` до довжини `width`, таким чином, щоб рядок був вирівняний за центром результуючого рядка. Якщо довжина рядка більша або рівна `width`, повертає оригінальний рядок.

```
>>> 'habr'.center(8, '_')
'__habr__'
>>> 'habr'.center(7, '_')
' _habr_ '
```

`str.zfill(width)`. Повертає копію рядка `str`, доповненого символами «0» до довжини `width`. Якщо рядок починається зі знаку («+» або «-»), то доповнюючі символи вставляються після знаку. Якщо довжина рядка більша або рівна `width`, повертає оригінальний рядок.

```
>>> '9'.zfill(4)
'0009'
>>> '-9'.zfill(4)
```

```
'-009'  
>>> '+9'.zfill(4)  
'+009'
```

str.format(*args, **kwargs). Повертає копію рядка *str*, відформатованого відповідним чином. Детальні відомості щодо цього методу викладено в додатку 3

7.4.7. Приклади розв'язування задач

Приклад. Написати програму за якою буде визначатися, чи є задане користувачем слово паліндромом (паліндром – слово, що читається однаково зліва направо і навпаки, наприклад, "шалаш").

```
s=input('Введіть рядок: ')  
if s==s[::-1]:  
    print('Паліндром')  
else:  
    print('Не паліндром')
```

Приклад. Дано рядок, який складається із слів розділених пропусками. Написати програму за якою буде обраховуватися кількість слів в рядку.

Якщо припустити, що пропуски містяться лише між словами їх там строго по одному, то розв'язок задачі може мати вигляд:

```
s=input('Введіть рядок: ')  
n=s.count(' ')+1  
print(n)
```

Якщо ж пропусків між словами може бути скільки завгодно, то розв'язок задачі може мати вигляд:

```
s=input('Введіть рядок: ')  
a=list(s.split(' '))  
b=[]  
for i in a:  
    if i!='':  
        b=b+[i]  
print(len(b))
```

Останній цикл можна переписати в вигляді генератору списку:

```
b =[i for i in a if i!='']
```

Приклад. Дано рядок. Написати програму за якою буде відбуватися поділ рядка на дві рівні частини (якщо довжина рядка непарна то перша частина має бути на 1 символ меншою) та виведення їх окремими рядками. При розв'язанні задачі не використовувати інструкцію if.

```
s=input('Введіть рядок: ')
print(s[:len(s)//2])
print(s[len(s)//2:])
```

Приклад. Дано рядок, який складається із слів розділених пропусками та розділовими знаками (,!.?). Написати програму за якою буде виведено всі слова окремими рядками.

```
s=input('Введіть рядок: ')
a=list(s.split(' '))
for i in a:
    if i!='':
        print(i.strip(',.!?'))
```

Останій цикл можна переписати в вигляді генератору списку:

```
[print(i.strip(',.!?')) for i in a if i!='']
```

Приклад. Дано рядок, який складається із слів розділених пропусками та розділовими знаками (,!.?). Написати програму за якою буде виведено всі слова парної довжини окремими рядками.

```
s=input('Введіть рядок: ')
a=list(s.split(' '))
a=[i.strip(',.!?') for i in a if i!='']
[print(i) for i in a if len(i)%2==0]
```

Приклад. Дано рядок, який складається із слів розділених пропусками та розділовими знаками (,!.?). Написати програму за якою буде виведено всі слова що не містять повторюваних символів.

```
s=input('Введіть рядок: ')
a=list(s.split(' '))
a=[i.strip(',.!?') for i in a if i!='']
for i in a:
    flag=True
    for j in i:
```



```
if i.count(j)!=1:
    flag=False
if flag:
    print(i)
```

7.5. Множини

Множина - це структура даних, що містить невпорядкований набір унікальних елементів. Множини в мові Python досить подібні до множин у математиці. Використання множин є доцільним у тому випадку, коли присутність елемента в наборі важливіша порядку слідування елементів та того, скільки разів даний елемент там зустрічається.

Множина може містити елементи різних типів, проте ці елементи можуть бути лише незмінюваних типів даних: числа, рядки, кортежі. Вимога незмінності елементів множини накладається особливостями подання множини в пам'яті комп'ютера.

Сама множина є змінюваним типом даних, тому до множин можна додавати нові та видаляти існуючі елементи. Як і у випадку математичних множин, у мові Python передбачено виконання операцій над множинами: об'єднання, перетину, різниці, симетричної різниці.

На відміну від масивів, де елементи зберігаються у вигляді послідовного списку, у множинах порядок зберігання елементів невизначений. Це дозволяє виконувати операції типу "перевірити приналежність елемента множині" швидше, ніж просто перебираючи всі елементи множини.

Множини записуються, як перелік елементів, розділених комою та взятих в фігурні дужки: {1, 2, 3, 'Hello'}.

7.5.1. Задання множини

Задання порожньої множини виконується з використанням функції `set()`:

```
>>> a = set()
```

Використання «{ }» призведе до створення порожнього словника:

```
>>> s={}
```

```
>>> type(s)
<class 'dict'>
```

Проте можна задати множину, перерахувавши її елементи, взяті в фігурні дужки «{}»:

```
>>> s1 = {1, 2, 3}
>>> s2 = {'a', 'b', 'c', 5}
```

Створення множини з інших структур даних

Множину можна отримати з елементів об'єкта, що може ітеруватися (діапазон, список, рядок, словник, кортеж, файл і т.д.), використавши функцію **set([iterable])**. Проте варто пам'ятати, що до множини будуть включені лише унікальні елементи.

```
>>> a = set('hello')
>>> a
{'e', 'o', 'l', 'h'}
>>> set(range(5))
{0, 1, 2, 3, 4}
```

7.5.2. Виконання дій над елементами множини

Як зазначалося раніше, можна перевірити приналежність деякого елемента до множини, використовуючи оператор **in** (значення **in** ім'я_множини).

```
>>> a = {1, 2, 3}
>>> 2 in a
True
>>> 2 not in a
False
```

Відповідно для перебору всіх елементів множини (в невизначеному порядку) необхідно скористатися циклом **for**:

```
s = {2, 5, 7, 11, 4}
for num in s:
    print(num)
```

7.5.3. Порівняння множин

Множини можна порівнювати між собою. Порівняння множин зводиться до перевірки, чи є множини рівними, або чи є певна множина підмножиною іншої.

```
>>> a={1,2,3}
>>> b={1,2,3}
>>> d={1,2,3,4}
>>> c={4,5,6}
```

set == other. Перевірка, чи множини `set` та `other` рівні. Повертає `True`, якщо всі елементи множини `set` належать множині `other`, і всі елементи множини `other` належать множині `set`, інакше – `False`.

```
>>> a == b
True
>>> a == d
False
```

set != other. Перевірка, чи є множини `set` та `other` не рівними. Повертає `True`, якщо принаймні один елемент множини `set` не належать множині `other`, або принаймні один елемент множини `other` не належать множині `set`, інакше – `False`.

```
>>> a != b
False
>>> a != d
True
```

set <= other. Перевірка, чи є множина `set` підмножиною множини `other`. Повертає `True`, якщо всі елементи множини `set` належать множині `other`, інакше – `False`.

```
>>> a<=d
True
>>> a<=b
True
```

set < other. Повертає `True`, якщо всі елементи множини `set` належать множині `other`, але не всі елементи множини `other` належать множині `set`, інакше – `False`.

```
>>> a<d
True
>>> a<b
False
```

set.isdisjoint(other). Повертає True, якщо множини set і other не мають спільних елементів, інакше – False.

```
>>> a.isdisjoint(c)
True
>>> c.isdisjoint(d)
False
```

set.issubset(other). Перевірка, чи є множина set підмножиною множини other. Аналогічно до set <= other.

set.issuperset(other). Перевірка чи є множина other підмножиною множини set. Аналогічно до set >= other.

7.5.4. Методи множин

set.add(x). Додає елемент x до множини set.

```
>>> s = {1, 2, 3}
>>> s.add(4)
>>> print(s)
{1, 2, 3, 4}
```

set.remove(x). Вилучає елемент x із множини set. Якщо такого елемента в множині немає, то виникає виняток KeyError.

set.discard(x). Вилучає елемент x із множини set. Якщо такого елемента в множині немає, то нічого не відбувається

set.pop(). Вилучає «перший» елемент з множини set та повертає його значення, як результат виконання функції. Так як множина – це неупорядкований набір, то не можна точно передбачити який з елементів буде взятий як перший. Якщо множина порожня, то виникає виняток KeyError.

set.clear(). Очищує множину set (вилучає всі елементи з множини).

Операції з множинами

```
>>> d={1,2,3,4}
```

```
>>> c={4,5,6}
```

set.union(*other) або **set | other | ...** Повертає об'єднання множин `set` і `other`. Множина-результат буде містити як елементи множини `set`, так і елементи множини `other`.

```
>>> d | c
```

```
{1, 2, 3, 4, 5, 6}
```

set.intersection(*other) або **set & other & ...** Повертає перетин множин `set` і `other`. Множина-результат буде містити елементи, які належать як множині `set`, так і множині `other`.

```
>>> d & c
```

```
{4}
```

set.difference(*other) або **set - other - ...** Повертає різницю множин `set` і `other`. Множина-результат буде містити елементи множини `set`, які не належать множині `other`.

```
>>> d - c
```

```
{1, 2, 3}
```

```
>>> c - d
```

```
{5, 6}
```

set.symmetric_difference(*other) або **set ^ other**. Повертає симетричну різницю множин `set` і `other`. Множина-результат буде містити елементи, які належать множинам `set` та `other`, але не належать обом множинам. Аналогічно до $(set | other) - (set \& other)$.

```
>>> d ^ c
```

```
{1, 2, 3, 5, 6}
```

Як можна бачити, вказані методи не призводять до зміни множини, а за ними відбувається формування нової множини. Проте для множин передбачені методи, що будуть призводити до зміни початкової множини.

set.update(*other) або **set |= other**. Додає до множини `set` всі елементи множини `other`. Множина `set` буде містити об'єднання множин `set` і `other`.

`set.intersection_update(*other)` або `set &= other.`

Вилучає з множини `set` всі елементи, які не входять до множини `other`.

Множина `set` буде містити перетин множин `set` і `other`.

`set.difference_update(*other)` або `set -= other.`

Вилучає з множини `set` всі елементи, які входять до множини `other`.

Множина `set` буде містити різницю множин `set` і `other`.

`set.symmetric_difference_update(other)` або

`set ^= other.` Множина `set` буде містити симетричну різницю множин `set` і `other`.

7.5.5. Приклади розв'язування задач

Приклад. Дано два рядки. Написати програму за якою буде створено дві множини (перша з символів першого рядка, друга з символів другого рядка) та виведено перетин, об'єднання, різницю цих множин.

```
st1=input('Введіть 1-й рядок')
st2=input('Введіть 2-й рядок')
s1=set(st1)
s2=set(st2)
o=s1|s2
p=s1&s2
r1=s1-s2
r2=s2-s1
print('Множина 1:',s1)
print('Множина 2:',s2)
print("Об'єднання 1:",o)
print('Перетин 1:',p)
print('Різниця між 1-ю і 2-ю множинами:',r1)
print('Різниця між 2-ю і 1-ю множинами:',r2)
```

Приклад. Дано рядок, який складається із слів розділених пропусками та розділовими знаками (.,!?). Написати програму за якою буде виведено всі слова що не містять повторюваних символів.

```

s=input('Введіть рядок: ')
a=list(s.split(' '))
a=[i.strip(',.!?') for i in a if i!='']
for i in a:
    if len(i)==len(set(i)):
        print(i)

```

Останій цикл можна переписати в вигляді генератору списку:

```
[print(i) for i in a if len(i)==len(set(i))]
```

8. Функції

У практиці програмування часто виникає ситуація, коли одну й ту саму групу операторів, які реалізують певну частину загальної задачі, треба повторити без змін в різних місцях програми. Для розв'язання цієї задачі використовується концепція підпрограм, яка отримала широке розповсюдження практично у всіх мовах програмування [7]. Так, наприклад, у мові Pascal підпрограми поділяються на процедури та функції, в мовах C та Python розглядаються лише функції. Також підпрограми (функції) використовуються для спрощення сприйняття коду за рахунок розбиття програми на менші, логічно завершені підпрограми.

Функції в Python практично нічим не відрізняються від функцій в інших мовах програмування. *Функцією* називають іменованій фрагмент програмного коду, до якого можна звернутися з іншого місця програми скільки завгодно разів [7] за її іменем. Єдиним виключенням, в Python, є lambda-функції, у яких немає імені. Тому в загальному можна зазначити, що функція - це така частина коду, яка ізольована від решти програми і виконуються тільки тоді, коли викликається.

Раніше йшлося про вбудовані функції мови Python: `sqrt()`, `len()`, `print()` та ін. Як можна було бачити, функції можуть приймати аргументи (нуль, один або кілька), і можуть повертати деякий результат (значення). Наприклад, функція `sqrt()` приймає один аргумент і повертає значення (корінь числа). Функція `print()` приймає змінне число аргументів і нічого не повертає.

При описі власних функцій потрібно уникати так званого побічного ефекту, тобто виконання функцією дій, відмінних від її основного призначення. Наприклад, у випадку обчислювального характеру функції не варто розміщувати в функції оператори виведення певних даних.

8.1. Опис та виклик функцій

Функція складається з заголовку та тіла функції. Заголовок функції відповідає за її опис, тобто іменування та вказання необхідних параметрів. Тіло ж функції відповідає за самі дії, які мають виконатися при виклику функції. Заголовок та тіло функції оформлюються, як основна та вкладена інструкції.

Заголовок функції починається зі службового слова `def`, після якого вказується ім'я функції та список її формальних параметрів у круглих дужках. Завершується заголовок функції символом двокрапка «:». З наступного рядка слідує тіло функції, яке може містити довільну кількість операторів, що записуються з однаковим відступом на початку рядків по відношенню до заголовку функції.

Формат опису функції:

```
def ім'я_функції(список_формальних_параметрів):  
    оператори_тіла_функції
```

Ім'я_функції – ідентифікатор, що формується за правилами створення ідентифікаторів і в подальшому буде використовуватися для виклику функції.

Формальні параметри – це назви змінних, що вказуються при описі функції і через які будуть передаватися дані з основної програми в функцію. Значення для формальних параметрів будуть вказуватися при виклику функції.

Список формальних параметрів записується у вигляді перерахованих через кому імен змінних. Як було зазначено раніше, функція може і не мати формальних параметрів.

Найпростіша функція, за якою виводиться привітання, буде мати наступний вигляд:

```
def func(name):  
    print('Hello', name)
```


У випадку, якщо тіло функції має єдиний оператор, як у нашому випадку, то функцію можна записати одним рядком:

```
def func(name): print('Hello', name)
```

Як можна бачити, ця функція має єдиний параметр `name`, який використовується як частина повідомлення для виведення на екран.

Для виклику функції вказується її ім'я з указаним в дужках списком фактичних параметрів.

Формат оператора виклику функції:

```
ім'я_функції([список фактичних параметрів]);
```

Виклик раніше описаної функції матиме вигляд:

```
>>> func('World')
Hello World
```

Фактичні параметри – це назви змінних чи конкретні значення, що вказуються при виклику функції.

Список фактичних параметрів записується у вигляді перерахованих через кому фактичних параметрів.

Формальні параметри інколи називають параметрами функції, фактичні параметри інколи називають аргументами функції. Відповідні фактичні та формальні параметри не обов'язково повинні мати однакові імена.

В загальному кількість фактичних параметрів повинна бути такою самою, як і кількість формальних параметрів. Це необхідно тому, що при виклику функції встановлюється взаємно однозначна відповідність між фактичними та формальними параметрами. Інколи ще говорять, що співставлення параметрів є позиційним, тобто значення фактичних параметрів присвоюються формальним параметрам відповідно до їх позиції. Проте в мові Python є особливості співставлення цих параметрів, які будуть розглянуті пізніше.

Наявність параметрів у функціях надає можливість програмісту робити функції більш гнучкими та універсальними. Параметри виступають вхідними чи допоміжними даними, які будуть використані в обчисленнях, передбачених функцією.

Для того, щоб функцією було повернуте певне значення, її тіло повинно містити інструкцію `return`. Інструкція `return` являє собою

службове слово `return`, після якого вказується значення чи змінна, значення якої потрібно повернути за функцією.

Наприклад, функція, за якою буде повернуто одиницю:

```
def one():  
    return 1
```

Якщо за функцією не передбачено повернення ніяких значень, функція може викликатися у головній програмі чи іншій функції як окремий оператор: `func('World')`.

Якщо за функцією передбачене повернення певного значення, то функція може викликатися у головній програмі чи іншій функції як операнд виразу: `two = one() + one()`.

Проте варто зауважити, що у випадку відсутності в тілі функції інструкції `return`, тобто не передбачення за функцією повернення ніякого значення, за функцією все рівно буде повернуте значення `None`.

```
>>> def func(): print('Hello!')  
>>> print(func())  
Hello!  
None
```

Іноколи виникає необхідність повернення за функцією не одного значення, а двох чи більше значень. Для цього потрібно в інструкції `return` вказати список або кортеж з бажаної кількості значень:

```
def f1(a, b): return [a**2, b**2]  
def f2(a, b): return (a**2, b**2)
```

Тоді виклик функції можна записати:

```
n1, m1 = f1(2, 3)  
n2, m2 = f2(2, 3)  
s1 = f1(2, 3)  
s2 = f2(2, 3)
```

В результаті виклику і виконання функцій отримаємо: `n1` та `n2` будуть містити значення 4, `m1` та `m2` будуть містити значення 9, `s1` буде містити список `[4, 9]`, `s2` буде містити кортеж `(4, 9)`.

В загальному, тіло функції може містити безліч інструкцій `return`, повернення значення за функцією буде визначатися першим викликом інструкції `return`.

Приклад. Написати функцію, за якою буде обчислюватися сума двох аргументів:

```
>>> def summa(a, b): return a+b
>>> summa(2, 3)
5
>>> x=5;
>>> y=6;
>>> summa(x, y)
11
```

Зважаючи на динамічну типізацію змінних в мові Python, аргументами однієї і тієї ж функції можуть бути значення чи змінні різного типу, наприклад:

```
>>> summa('Hello ', 'world.')
'Hello world.'
>>> summa([2, 3, 4], [4, 5, 6])
[2, 3, 4, 4, 5, 6]
```

Проте при такому багатогранному використанні функцій варто пам'ятати про строгу типізацію мови Python і неможливість проведення операцій у виразах з даними різних несумісних типів. Так, наступний виклик функції `summa('Hello ',5)` призведе до виникнення винятку `TypeError (unsupported operand type(s) for +: 'int' and 'str')`.

8.2. Розширене використання параметрів та аргументів

В мові Python передбачена досить гнучка можливість використання параметрів, а саме: можливість вказання для параметрів значень за замовчуванням; використання параметрів, до яких можна звернутися за їх назвою (ключових аргументів); створення функцій, що будуть приймати змінну кількість аргументів; вказання обов'язковості ключових аргументів.

8.2.1. Значення параметрів за замовчуванням

При описі функцій інколи виникають випадки, що деякий параметр при виклику функції в більшості випадків буде мати одне й те саме значення, але незважаючи на це, все ж можуть бути випадки виклику цієї ж функції, з

іншим значенням даного параметру. Наприклад, функція `print()` має параметр `sep`, який досить часто не вказується при її виклику, а значенням для нього в такому випадку встановлюється символ пропуску. Таке значення для параметру називається значенням за замовчуванням, а сам параметр стає необов'язковим.

Для вказання того, що параметр матиме значення за замовчуванням, необхідно при описі функції після імені цього параметру поставити знак присвоєння (`=`) і вказати відповідне значення. Значення за замовчуванням має бути незмінюваним.

Наприклад:

```
>>> def summa(a, b=2): return a+b
>>> summa(4, 6)
10
>>> summa(4)
6
```

В першому випадку при виклику функції вказані обидва аргументи (фактичні параметри), тому параметру `a` буде надане значення 4, параметру `b` – 6 (сума рівна 10). В другому випадку виклик функції містить лише один аргумент, тому параметру `a` буде надане значення 4 (це перший параметр, а відповідно йому надається значення першого аргументу), другому ж параметру буде надане значення 2, яке є його значенням за замовчуванням.

Параметрів зі значенням за замовчуванням у списку параметрів може бути будь-яка кількість. При виклику функції значення вказаних аргументів будуть надаватися параметрам функції послідовно. Це пов'язано з тим, що за замовчуванням аргументи є позиційними, тобто значення аргументів присвоюються параметрам відповідно до їх позиції.

```
>>> def func(a, b=2, c=10):
    print('a=', a, 'b=', b, 'c=', c)
>>> func(3, 4, 5)
a= 3 b= 4 c= 5
>>> func(3, 4)
a= 3 b= 4 c= 10
>>> func(3)
a= 3 b= 2 c= 10
```

Параметри зі значенням за замовчуванням у списку параметрів не можуть передувати параметрам без значення за замовчуванням. Тобто значеннями за замовчуванням можуть бути забезпечені тільки параметри, які знаходяться в кінці списку параметрів. Наприклад, припустимо, що є функція, описана наступним чином: `def summa (a=2, b): return a+b`. При її виклику `summa(3)`, незважаючи на те, що параметр `a` має значення за замовчуванням, йому буде надане значення першого аргументу (тобто, 3), а для параметру `b` аргументу не вистачить, відповідно функція не зможе бути викликана.

8.2.2. Ключові аргументи

Окрім позиційних аргументів, коли значення параметрів встановлюються у відповідності зі слідуванням аргументів, у мові Python передбачене використання так званих ключових аргументів. Ключовий аргумент – це аргумент, який записується з указанням імені параметра, для якого він призначений ("ім'я_параметра = значення").

Наприклад, є деяка функція з параметрами, що мають значення за замовчуванням. А при виклику цієї функції необхідно вказати значення тільки для деяких параметрів. В такому випадку якраз і стане у нагоді використання ключових аргументів. При виклику функції можна буде вказати лише значення для тих параметрів, яким не підходять їх значення за замовчуванням.

Для задання ключового аргументу необхідно вказати ім'я параметра та після знаку присвоєння необхідне значення.

```
>>> def func(a, b=2, c=10):  
    print('a=', a, 'b=', b, 'c=', c)  
>>> func(3, c=15)  
a= 3 b= 2 c= 15  
>>> func(3, c=15, b=8)  
a= 3 b= 8 c= 15
```

Аргумент може бути ключовим не лише для параметра зі значенням за замовчуванням. Проте в такому випадку такий ключовий аргумент має бути обов'язково присутній при виклику функції.

```
>>> func(c=22, a=23)
a= 23 b= 2 c= 22
```

Можна виділити декілька переваг використання ключових аргументів: по-перше, використання функції стає легшим, оскільки немає необхідності відстежувати порядок аргументів; по-друге, можна задавати значення тільки деяким параметрам, за умови, що решта параметрів мають значення за замовчуванням.

Проте в версії 3.8 з'явилася можливість обмеження, щодо використання ключових аргументів. Якщо між параметрами функції розмістити символ «/» то параметри які йдуть до нього будуть лише позиційними, а параметри які йдуть за ним можуть бути ключовими.

```
>>> def func(a, b=2, /, c=10):
    print('a=', a, 'b=', b, 'c=', c)
>>> func(3, 4, 5)
a= 3 b= 4 c= 5
>>> func(3, 4, c=5)
a= 3 b= 4 c= 5
>>> func(3, b=4, c=5)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    func(3, b=4, c=5)
TypeError: func() got some positional-only arguments
passed as keyword arguments: 'b'
```

8.2.3. Змінна кількість аргументів

Іноді буває необхідно визначити функцію, що може бути викликана з будь-якою кількістю аргументів, тобто здатну приймати змінну кількість аргументів. Наприклад, функція `print()` може приймати декілька (одне, два, три і т.д.) значень, що мають бути виведені, чи функція `min()` може приймати декілька (два, три і т.д.) значень для визначення мінімального з них. Така можливість реалізується за рахунок того, що під час виклику функції змінна кількість аргументів буде зібрана (упакована) в одну змінну типу кортеж або словник та передана в функцію.

Змінна кількість аргументів може застосовуватися і для позиційних, і для ключових аргументів. Окрім того, при описі функції можуть бути присутні параметри як для одиничних аргументів, так і для змінної кількості аргументів, головне щоб опис параметрів для одиничних аргументів передували параметрам для змінної кількості аргументів.

Для вказання того, що функція зможе приймати змінну кількість позиційних аргументів, при її описі необхідно вказати параметр, перед яким поставити знак «*». Наприклад, для функції:

```
def summa(first=0, *param):
    print('first=', first, '; param=', param, sep='')
    s=first
    for number in param:
        s+=number
    return s
```

можливі різні варіанти її виклику:

```
>>> print('summa=', summa())
first=0; param=()
summa= 0
>>> print('summa=', summa(5,3,4))
first=5; param=(3, 4)
summa= 12
>>> print('summa=', summa(5,3,4,5,6,7,8))
first=5; param=(3, 4, 5, 6, 7, 8)
summa= 38
```

Отже, можна бачити, що всі позиційні аргументи, починаючи з другого, збираються в кортеж під ім'ям param.

Для вказання того, що функція зможе приймати змінну кількість ключових аргументів, при її описі необхідно вказати параметр, перед яким поставити два знаки «**». Наприклад, для функції:

```
def summa(first=0, **param):
    print('first=', first, '; param=', param, sep='')
    s=first
    for key in param:
        s+=param[key]
```

```
    return s
```

можливі різні варіанти її виклику:

```
>>> print('summa=', summa(2, s1=3, s2=4))
first=2; param={'s1': 3, 's2': 4}
summa= 9
>>> print('summa=', summa(s1=3, s2=4))
first=0; param={'s1': 3, 's2': 4}
summa= 7
```

Отже, можна бачити, що всі ключові аргументи, починаючи з другого, збираються в словник під ім'ям `param`.

Розглянуті випадки можуть бути використані одночасно:

```
def summa(first=0, *param1, **param2):
    print('first=', first)
    print('param1=', param1)
    print('param2=', param2)
    s=first
    for number in param1:
        s+=number
    for key in param2:
        s+=param2[key]
    return s
>>> print('summa=', summa(3,4,5,6))
first= 3
param1= (4, 5, 6)
param2= {}
summa= 18
>>> print('summa=', summa(s1=3, s2=4))
first= 0
param1= ()
param2= {'s1': 3, 's2': 4}
summa= 7
>>> print('summa=', summa(3,4,5,6,s1=3, s2=4))
first= 3
param1= (4, 5, 6)
```



```
param2= {'s1': 3, 's2': 4}
summa= 25
```

8.2.4. Обов'язкові ключові аргументи

Іноколи є необхідність опису функцій, в яких певні параметри мають бути доступні лише за ключовими аргументами, що забезпечить більшу зрозумілість її параметрів. Наприклад, функція `print()` має параметри `sep` та `end`, вказати значення для яких можна лише з використанням ключових аргументів.

Для визначення параметра, який зможе приймати значення тільки від ключового аргументу, його необхідно оголосити після параметра з зірочкою (параметра, що буде приймати змінну кількість аргументів).

```
def summa(first=0, *param, mult):
    print('first=', first, '; param=', param, sep='')
    print(mult=' ', mult)
    s=first
    for number in param:
        s+=number
    s*=mult
    return s

>>> print('result=', summa(1,2,3, mult=5))
first=1; param=(2, 3)
mult= 5
result= 30

>>> print('result=', summa(1,2,3,5))
```

У разі виконання оператора `print('result=', summa(1,2,3,5))`, отримаємо помилку, оскільки параметра `mult` не буде надане значення, так як всі аргументи, починаючи з другого, будуть зібрані в кортежі `param`.

Якщо за функцією не передбачається наявність змінної кількості аргументів, але є необхідність в параметрі лише за ключовими аргументами, то заголовок такої функції можна визначити так:

```
def summa(first=0, *, mnog):
```

8.3. Глобальні та локальні змінні

Кожна змінна має свою область видимості. Область видимості всіх змінних обмежена блоком, в якому вони оголошені, починаючи з точки оголошення. Тому при введенні у програму функції виникає поділ змінних і відповідних їм даних на глобальні і локальні.

8.3.1. Глобальні змінні

Всередині функції можуть бути використані змінні, оголошені в основній програмі. Наприклад, маємо програму:

```
def f():  
    print(Z)  
Z = 1  
f()
```

В результаті виконання такої програми буде виведене число 1.

Під час запуску програми функція `f()` буде інтерпретуватися лише після її виклику. Проте до виклику функції `f()` змінній `Z` присвоюється значення 1. Отже, коли дійде черга до виконання функції, змінна `Z` вже буде ініціалізована, а отже за оператором `print(Z)` буде виведене її значення.

Змінні, оголошені за межами функції, але доступні всередині функції, називаються глобальними.

8.3.2. Локальні змінні

Якщо в тілі функції ініціалізувати деяку змінну, то її областю видимості буде тіло цієї функції. Отже, використання такої змінної за межами функції недопустиме. Наприклад, маємо програму:

```
def f():  
    Z = 1  
f()  
print(Z)
```

В результаті виконання такої програми отримаємо виняток: `NameError: name 'Z' is not defined`.

Змінні, оголошені всередині функції або вказані у списку формальних параметрів функції, називаються локальними. За межами функції її локальні змінні недоступні.

Обмежена доступність локальних змінних надає можливість використовувати змінні з одним і тим же ім'ям в різних функціях.

8.3.3. Зв'язок однойменних локальних і глобальних змінних

Під час написання програми може виникнути випадок, коли програма матиме локальні та глобальні змінні з однаковими іменами. В такому випадку може виникнути питання: що ж буде відбуватися з локальними та глобальними змінними під час зміни їх значень?

Наприклад, маємо програму:

```
def func():
    Z=2
    print('Z всередині функції =', Z)
Z=5
func()
print('Z за межами функції=', Z)
```

Після її виконання отримаємо:

```
Z всередині функції= 2
Z за межами функції= 5
```

Незважаючи на те, що значення змінної Z змінилося всередині функції, за межами функції воно залишилося незміненим. Це пов'язане з тим, що при виконанні функції `func()` було ініціалізовано локальну змінну Z, а отже областю видимості даної змінної стало лише тіло функції. Надане значення локальній змінній Z ніяк не вплинуло на значення глобальної змінної Z. Така реалізація забезпечує "захист" глобальних змінних від випадкової зміни в тілі функції.

Якщо всередині функції модифікується значення деякої змінної, то вона стає локальною, і її модифікація не призводить до зміни глобальної змінної з таким же ім'ям. Більш формально можна це сформулювати так: інтерпретатор Python вважає змінну локальною для даної функції, якщо в тілі функції є хоча б одна інструкція, що модифікує значення цієї змінної.

Інструкцією, що модифікує значення змінної, може бути оператор присвоєння «`=`», або використання змінної в якості параметра циклу `for`.

Зміна значень глобальних змінних у функції

Іноколи виникає необхідність написання функцій, в середині яких зміна значень глобальних змінних є необхідною. В такому випадку для забезпечення можливості зміни значення глобальної змінної в середині функції необхідно оголосити цю змінну в тілі функції за допомогою ключового слова `global`:

```
def func():
    global Z
    print('Z в середині функції (до зміни) =', Z)
    Z=2
    print('Z в середині функції (після зміни) =', Z)
Z=5
func()
print('Z за межами функції =', Z)
```

Після виконання отримаємо:

```
Z в середині функції (до зміни) = 5
Z в середині функції (після зміни) = 2
Z за межами функції = 2
```

Проте, краще не виконувати зміну значень глобальних змінних всередині функції. Якщо за функцією необхідно змінити значення змінної, краще повернути це значення функцією, і в головній програмі виконати операцію зміни. Якщо слідувати цьому правилу, то функції виходять незалежними від коду головної програми, і їх можна легко копіювати з однієї програми в іншу.

8.3.4. Нелокальні змінні

Окрім локальних та глобальних змінних є ще один тип змінних, так звана «нелокальна» (`nonlocal`) змінна. Нелокальна змінна є чимось середнім між локальною та глобальною. Такі змінні зустрічаються в функціях, що визначені в середині інших функцій. Для оголошення нелокальної змінної використовується службове слово `nonlocal`.

```

def func():
    Z=2
    print('Z в середині функції (до зміни) =', Z)
    def func1():
        nonlocal Z
        Z=3
    func1()
    print('Z в середині функції (після зміни) =', Z)
Z=5
func()
print('Z за межами функції =', Z)

```

Опис нелокальної змінної `nonlocal Z` всередині функції `func1()` означає, що областю видимості цієї змінної є не тіло функції `func1()`, а тіло функції `func()`.

Після виконання отримаємо:

```

Z в середині функції (до зміни) = 2
Z в середині функції (після зміни) = 3
Z за межами функції = 5

```

Якщо ж в тілі функції `func1()` буде відсутній оператор `nonlocal Z`, то матимемо: глобальну змінну `Z`, локальну змінну `Z` для функції `func()` та локальну змінну `Z` для функції `func1()`, тобто три окремі незалежні змінні.

8.4. Правила локалізації

Програма на Python має модульну структуру і може складатися з ряду вкладених один в одного блоків (функцій). Головна програма – це найбільший блок. Змінні, описані в головній програмі, є глобальними і можуть використовуватись у всіх вкладених блоках. Змінні, описані в вкладеному блоці, є локальними і недоступні у зовнішніх блоках.

Для правильного використання змінних слід дотримуватись таких правил щодо їх ідентифікаторів:

- Кожний ідентифікатор має бути ініціалізований до його використання.
- Областю дії ідентифікатора є блок, в якому його ініціалізовано.

- Всі ідентифікатори в блоці мають бути унікальними, тобто не повторюватися.
- Один і той самий ідентифікатор може бути по-різному ініціалізований в кожному окремому блоці.

8.5. Lambda функції

Як було зазначено раніше, за допомогою ключового слова `def` можна описати іменовану функцію, а потім, використовуючи ім'я функції, викликати її на виконання з довільного місця програми. Також зазначалося, що в мові Python передбачена можливість використання `lambda`-функцій, у яких немає імені. Такі функції інколи називають анонімними. `lambda`-функція може містити тільки один оператор, який завжди повертає об'єкт, який можна присвоїти змінній. Згодом ця змінна може бути використана для того, щоб звернутися до функції (зворотний виклик) в будь-якому місці програми.

Таким чином, використання `lambda`-функцій надає можливість програмістам мати альтернативний синтаксис для створення функцій.

Наприклад функцію: `def sqrt(x): return x ** 0.5`
можна записати так: `sqrt = lambda x: x ** 0.5`

В обох випадках виклик `sqrt(25)` поверне результат 5.0.

`lambda`-функції часто використовуються для вбудовування функції в будь-яке місце коду. Прикладом використання `lambda`-функцій є опрацювання списків всередині функцій `map()`, `filter()`, `reduce()`. Наприклад, функція `map` приймає два аргументи, перший - це функція, яка буде застосована до кожного елемента списку, а другий - це список, який потрібно опрацювати.

```
>>> m=[1,2,3,4,5,6,7]
>>> list(map(lambda x: x*3, m))
[3, 6, 9, 12, 15, 18, 21]
```

Іншим прикладом використання `lambda`-функцій є вбудовування таких функцій безпосередньо в список аргументів:

```
>>> funcs = [lambda x : x**2, lambda x : x**3,
             lambda x : x**4]
>>> for func in funcs : print('Result:', func(3))
```

```
Result: 9
Result: 27
Result: 81
```

8.5. Рекурсія

Як ми бачили вище, функція може викликати іншу функцію. Але функція також може викликати і саму себе. Під рекурсією розуміють виклик функції з тіла цієї ж самої функції. Функцію, яка містить рекурсію, називають рекурсивною.

Розглянемо реалізацію рекурсії на прикладі функції обчислення факторіала та обчислення цілого степеня числа.

Формула обчислення факторіала:

$n! = 1$, якщо $n = 0$;

$n! = n * (n-1)!$, якщо $n > 0$.

Формула обчислення цілого степеня числа:

$x^n = 1$, якщо $n = 0$;

$x^n = x * x^{n-1}$, якщо $n > 0$.

З формул видно, що для обчислення кожного наступного значення треба знати попереднє. Розглянемо реалізацію функцій обчислення факторіала і цілого степеня числа.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
def intPower(x, n):
    if n == 0:
        return 1
    else:
        return x * intPower(x, n-1)
```

Рекурсивні функції є потужним механізмом у програмуванні. На жаль, вони не завжди ефективні стосовно пам'яті. Це пов'язане з тим, що кожний "вкладений" виклик функції – це черговий набір всіх локальних змінних цієї функції.

Також неуважність при описі рекурсивної функції може привести до нескінченості рекурсії, коли ланцюжок викликів функцій ніколи не завершується і триває, поки не скінчиться вільна пам'ять в комп'ютері. Тому необхідністю для працездатності рекурсивних функцій є наявність правильно оформленої умови закінчення рекурсивних викликів (наприклад, перевірка значення параметра, що змінюється).

8.6. Приклади розв'язування задач

Приклад. Написати програму, за якою буде обчислюватися площа трикутника, заданого довжинами його сторін.

```
import math
def geron(a,b,c):
    p=(a+b+c)/2
    return math.sqrt(p*(p-a)*(p-b)*(p-c))
print('Введіть довжини сторін трикутника:')
a=float(input())
b=float(input())
c=float(input())
if a+b>c and a+c>b and b+c>a:
    s=geron(a,b,c)
    print('Площа трикутника =', s)
else:
    print('Трикутник з даними сторонами не існує.')
```

Приклад. Обчислити значення виразу

$$f(x)=x^2+x+3+ex^2+x+3+\sin^2(x^2+x+3) \quad x=1,2,\dots,10.$$

```
import math
def y(x):
    return x*x+x+3
for i in range(1,11):
    f=y(i)+ math.pow(math.e, y(i)) +
    math.pow(math.sin(y(i)), 2)
    print('f({})={}'.format(i, f))
```

Приклад. Знайти довжину вектора, заданого своїми координатами у n -вимірному просторі.


```

import math
def vLength(v, n):
    s=0
    for i in range(n):
        s+=v[i]**2;
    return math.sqrt(s)
n=int(input('Введіть кількість елементів списку = '))
a=[]
for i in range(1,n+1):
    x=int(input('a[{}]='.format(i)))
    a.append(x)
print('Довжина вектора a=',vLength(a,n));

```

Приклад. Дано a,b,c – довжини сторін деякого трикутника. Знайти медіани трикутника.

Довжина медіани, проведеної до сторони a, дорівнює

$$m_a = \frac{1}{2} \sqrt{2b^2 + 2c^2 - a^2}$$

```

import math
def median(x, y, z):
    return 0.5*math.sqrt(2*x*x+2*y*y-z*z)
print('Введіть довжини сторін трикутника:')
a=float(input())
b=float(input())
c=float(input())
if a+b>c and a+c>b and b+c>a:
    m1=median(a, b, c)
    m2=median(a, c, b)
    m3=median(c, b, a)
    print('Медіана 1 =', m1)
    print('Медіана 2 =', m2)
    print('Медіана 3 =', m3)
else:
    print('Трикутник з даними сторонами не існує.')

```

Приклад. Дано цілочисельну матрицю випадкових цілих чисел розміром $m \times n$ (m і n задається користувачем). Знайти мінімальний елемент матриці. Описати функції генерування матриці, виведення матриці та функцію пошуку мінімального елемента.

```
import random
def gener_matr(n,m):
    a=[]
    for i in range(n):
        b=[]
        for j in range(m):
            x=random.randint(10,99)
            b.append(x)
        a.append(b)
    return a
def print_matr(a):
    print('Згенерована матриця:')
    for row in a:
        for elem in row:
            print(elem, end=' ')
        print()
def min_matr(a):
    min_col=list(map(min, a))
    min_el=min(min_col)
    return min_el

n=int(input('Введіть кількість рядків матриці: '))
m=int(input('Введіть кількість стовпців матриці: '))
a=gener_matr(n,m)
print_matr(a)
print('Мінімальний елемент: ',min_matr(a))
```

9. Файли

Досить часто при створенні програм виникає необхідність у використанні даних, які записані на зовнішніх носіях (дисках) і оформлені у

вигляді файлів даних. Файл може бути джерелом інформації – тоді відбувається читання з файла, або приймачем – тоді відбувається запис даних у файл [7].

Для роботи з файлом необхідно пов'язати його логічне позначення (файлову змінну) з фізичним файлом (файл, який зберігається на диску).

9.1. Відкриття та закриття файлу

Для відкриття файлу використовується функція `open()`, за якою повертається файловий об'єкт. Виклик функції матиме вигляд:

```
file=open()
```

В результаті змінна `file` буде містити файловий об'єкт, який пов'язаний з відповідним файлом на диску (ім'я якого та режими відкриття вказуються параметрами функції). Для виконання операцій з цим файлом необхідно буде викликати методи для змінної `file`.

Повний формат цієї функції має наступний вигляд

```
open(fname, mode='r', buffering=-1, encoding=None,  
errors=None, newline=None, closefd=True, opener=None)
```

`fname` - цілочисельний файловий дескриптор або абсолютний чи відносний шлях до файлу, який потрібно відкрити.

`mode` - рядковий параметр за яким вказується режим відкриття файлу. Файл може бути відкритий в таких режимах: 'r' - відкрити файл для читання (є значенням за замовчуванням); 'w' - відкрити файл для запису. Якщо файлу не існує, то створюється новий файл, якщо файл існує, то вміст файлу вилучається; 'x' - відкрити файл для запису. Якщо файлу не існує, то створюється новий, якщо файл існує, то виникає виняток `FileExistsError`; 'a' - відкрити файл для до запису, який відбувається в кінець файлу; '+' - відкрити файл для читання та для запису, який відбувається в кінець файлу (має суттєве значення лише в поєднанні з 'r', тобто 'r+', проте може використовуватися в поєднанні і з іншими режимами)

Для будь-яких з перерахованих вище режимів додатково може бути вказаний символ 'b' або 't' (є значенням за замовчуванням), за яким буде розрізнятися двійковий та текстовий ввід/вивід (наприклад, 'rb', 'wt' і т. д.). Файл, відкритий в двійковому режимі, повертає вміст у вигляді об'єктів

байт без будь-якого декодування В текстовому режимі – вміст файлу повертається як `str`.

`buffering` – параметр, за яким вказується політика буферизації при роботі з файлом. Може набувати таких значень: 0 - використовується лише для двійкового режиму і відключає буферизацію; 1 - використовується лише для текстового режиму і встановлює буферизацію рядка; ціле додатне число - визначає розмір буферу в байтах..

`encoding` – параметр, за яким вказується кодування файлу (використовується лише в текстовому режимі). Кодування за замовчуванням залежить від платформи. Поточне кодування можна отримати за методом `locale.getpreferredencoding()`.

`errors` – параметр, за яким вказується метод опрацювання помилок при кодуванні та декодуванні файлу (використовується лише в текстовому режимі). Можливі значення: `None`, `'strict'`, `'ignore'`, `'replace'`, `'surrogateescape'`, `'xmlcharrefreplace'`, `'backslashreplace'`, `'namereplace'` та інші.

`newline` – параметр, за яким вказується опрацювання вказівника закінчення рядка (використовується лише для текстових файлів). Можливі значення: `None`, `'`, `'\n'`, `'\r'`, `'\r\n'`. *Читання з файлу*: при значенні `newline=None` після читання вказівником закінчення рядка буде встановлений `'\n'`; при інших значеннях параметра — вказівник закінчення рядка залишиться без змін. *Запис до файлу*: при значенні `newline=None` в кінці кожного рядка буде встановлений вказівник закінчення рядка за замовчуванням системи (`os.linesep`); якщо `newline='` або `newline='\n'`, то вказівником закінчення рядка буде `'\n'`; інші значення параметра явно вказують вказівник закінчення рядка.

`closefd` – параметр, за яким вказується необхідність закриття файлового дескриптора (використовується лише для файлів, вказаних своїм дескриптором). При значенні параметра `closefd=False` файловий дескриптор залишиться відкритий навіть після явного закриття файлу, інакше файловий дескриптор закривається разом з закриттям файлу.

`opener` – параметр, за яким може бути вказаний користувацький метод відкриття файлу.

Проте на практиці найчастіше використовуються параметри `fname` та `mode`.

Після роботи з файлом його обов'язково необхідно закрити, для цього передбачений метод `close()`.

9.2. Атрибути файлового об'єкта

У файлового об'єкта є ряд атрибутів, які можна отримати навіть після закриття файлу.

`file.closed` – містить `True`, якщо файл закритий і `False`, якщо файл відкритий.

`file.mode` – містить режим доступу до файлу

`file.name` – містить ім'я файлу

```
>>> f=open("test.txt", "r")
>>> print("file.closed: " + str(f.closed))
file.closed: False
>>> print("file.mode: " + f.mode)
file.mode: r
>>> print("file.name: " + f.name)
file.name: test.txt
>>> f.close()
>>> print("file.closed: " + str(f.closed))
file.closed: True
```

Також для файлового об'єкта передбачені два методи, за якими визначається можливість запису даних до файлу (**`file.readable()`**) чи можливість зчитування даних з файлу (**`file.writable()`**). Результатом цих методів є значення логічного типу.

9.3. Читання з файлу

Для зручності будемо вважати, що на диску вже є текстовий файл `test.txt`, який має наступну структуру:

```
1 2 3 4 5
Work with file
```

Для зчитування даних з файлу передбачено кілька методів.

file.read(size). Зчитує з файлу `file` `size` символів. Якщо параметр `size` не заданий чи його значення є від'ємним, то зчитується весь файл. Якщо досягнуто кінець файлу, то за методом `file.read()` буде повернуто порожній рядок ".

```
>>> f=open("test.txt", "r")
>>> f.read()
'1 2 3 4 5\nWork with file\n'
>>> f.close()
>>> f=open("test.txt", "r")
>>> f.read(5)
'1 2 3'
>>> f.close()
```

За методом **file.readline()** зчитується один рядок з файлу `file`. Символ закінчення рядка “\n” залишається в кінці рядка і відсутній лише в останньому рядку, якщо файл не закінчується порожнім рядком. Це робить однозначним значення, отримане за методом `readline()`; якщо отримано порожній рядок, то було досягнуто кінець файлу, порожній же рядок подається символом “\n”.

```
>>> f=open("test.txt", "r")
>>> f.readline()
'1 2 3 4 5\n'
>>> f.close()
```

Порядкове читання з файлу можна виконати, використовуючи оператор `for`, тобто можна організувати перебір всіх елементів з файлу.

```
>>> f=open("test.txt", "r")
>>> for line in f: print(line, end='')
1 2 3 4 5
Work with file
>>> f.close()
```

За необхідності прочитати всі рядки в список, можна скористатися методами: **list(file)** або **file.readlines()**

```
>>> f=open("test.txt", "r")
>>> a=list(f)
>>> a
```

```

['1 2 3 4 5\n', 'Work with file']
>>> f.close()
>>> f=open("test.txt", "r")
>>> b=f.readlines()
>>> b
['1 2 3 4 5\n', 'Work with file']
>>> f.close()

```

9.4. Запис у файл

Для запису даних у файл використовується метод **file.write(string)**.

При вдалому запису рядка до файлу за методом повертається кількість записаних символів.

```

>>> f=open("test.txt", "a")
>>> f.write("Test string")
11
>>> f.close()

```

Щоб записати щось відмінне від рядка, воно повинно бути приведенне до рядка:

```

>>> f=open("test.txt", "a")
>>> f.write(str(2019))
4
>>> f.close()

```

Варто відмітити, що за методом `file.write(string)` відбувається запис відповідних рядкових даних у файл, і ніякого додавання вказівника кінця рядка не відбувається. Тобто всі дані заносяться послідовно, без розбиття на рядки, якщо це явно не вказується.

Якщо поглянути на вміст файлу "test.txt", то можна побачити наступне:

```

1 2 3 4 5
Work with file2019

```

Запис даних до файлу можна також організувати з використанням методу `print()`. Як зазначалося раніше, цей метод має необов'язковий

параметр `file`, який за замовчуванням рівний `sys.stdout` (стандартний пристрій виведення - екран).

Якщо ж вказати значенням для цього параметра файлоу змінну, то за методом `print()` буде відбуватися виведення даних не на екран, а в файл (запис даних до файлу).

```
>>> f=open("test.txt", "a")
>>> print("Test string1", file=f)
>>> print("Test string2", file=f)
>>> f.close()
```

На відміну від методу `file.write(string)`, за методом `print()` крім запису власне рядкових даних відбувається і дописування вказівника кінця рядка.

Якщо поглянути на вміст файлу `"test.txt"`, то можна побачити наступне:

```
1 2 3 4 5
Work with file2019Test string1
Test string2
```

9.5 Додаткові методи роботи з файлами

`file.tell()`. Повертає ціле число, яке відповідає позиції файлового покажчика («умовного курсору») в файлі. Визначається в байтах (символах).

Наприклад, якщо прочитати перші п'ять символів, то файловий покажчик буде встановлений на позиції 5.

```
>>> f=open("test.txt", "r")
>>> f.read(5)
'1 2 3'
>>> f.tell()
5
>>> f.close()
```

`file.seek(offset, from_what=0)`. Встановлює файловий покажчик в нову позицію, яка обчислюється додаванням зміщення `offset` до точки відліку заданої параметром `from_what`. При `from_what=0` –

точкою відліку є початок файлу; при `from_what=1` – точкою відліку є поточна позиція файлового покажчика; при `from_what=2` - точкою відліку є кінець файлу. Значення параметра `from_what`, відмінне від 0, використовується лише для бінарних файлів.

```
>>> f=open("test.txt", "rb")
>>> f.seek(4)
>>> f.read(1)
b'3'
>>> f.seek(1, 1)
>>> f.read(1)
b'4'
>>> f.close()
```

9.6. Використання менеджера контексту

Для деяких об'єктів визначені стандартні завершальні дії, які повинні бути виконані після завершення роботи з цим об'єктом. Причому такі завершальні дії повинні виконуватися незалежно від того, чи пройшли операції з об'єктом успішно, чи виникла помилка. Звичайно, можна скористатися блоком опрацювання виняткових ситуацій, проте в мові Python передбачений менеджер контексту `with as`.

Розглянемо можливості використання менеджера контексту при роботі з файлами. Послідовність роботи з файлом складається з таких кроків: відкрити файл, опрацювати (прочитати чи записати) дані файлу, закрити файл. Зрозуміло, що при опрацюванні файлу можуть виникнути нештатні ситуації, які призведуть до завершення програми, окрім того, обов'язково слід потурбуватися про закриття файлу. Використання оператора `with as` при роботі з файлами буде забезпечувати виконання завершальних дій без явного їх вказування, тобто закриття файлу.

```
with open("test.txt") as f:
    for line in f:
        print(line, end="")
```

Після завершення виконання зазначеного коду файл `f` завжди закривається, навіть якщо виникла проблема при обробці файлу. Отже, наведений код не потребує виклику оператора закриття файлу.

9.7. Приклади розв'язування задач

Приклад. Написати програму створення текстового файлу з рядків заданих користувачем.

```
fn=input("Введіть ім'я файлу: ")
f=open(fn, 'w')
print('Введіть рядки для запису в файл.')
print('Для завершення введіть "end." без лапок.')
s=input()
while s!='end.':
    f.write(s+'\n')
    s=input()
f.close()
```

Приклад. Дано текстовий файл, який містить дані записані окремими рядками. Вивести на екран всі рядки файлу згрупувавши спочатку рядки парної довжини, а потім рядки непарної довжини.

```
fn=input("Введіть ім'я файлу: ")
with open(fn, 'r') as f:
    print('Рядки парної довжини:')
    for i in f:
        if (len(i)-1)%2==0: print(i, end='')
    print('Рядки не парної довжини:')
    f.seek(0)
    for i in f:
        if (len(i)-1)%2==1: print(i, end='')
```

Приклад. Дано текстовий файл кожен рядок якого містить координату точок в декартовій системі координат. Координата кожної точки записана двома дійсними числа через пропуск. Вивести на екран координату точки максимально віддаленої від початку координат та відстань до неї.

```
dist=0
dist_x=0
```

```

dist_y=0
import math as m
fn=input("Введіть ім'я файлу: ")
with open(fn, 'r') as f:
    for i in f:
        x, y=i.split()
        x, y=float(x), float(y)
        tmp=m.sqrt(x**2+y**2)
        if tmp>dist:
            dist=tmp
            dist_x=x
            dist_y=y
print('На максимальній відстані ({:.3}) від початку
координат знаходиться точка з координатами ({}
, {})' .format(dist,dist_x,dist_y))

```

Приклад. Дано цілочисельну матрицю випадкових цілих чисел розміром $n \times n$ (n задаються користувачем). Написати програму для запису даної матрицю до файлу в «природному» вигляді (кожен окремий рядок файлу має містити елементи відповідного рядка матриці записані через пропуск).

```

import random as rnd
n=int(input('Вкажіть розмірність матриці:'))
m=[]
for i in range(n):
    a=[]
    for j in range(n):
        x=rnd.randint(1,9)
        a.append(x)
    m.append(a)
fn=input("Введіть ім'я файлу: ")
with open(fn,'w') as f:
    for i in m:
        for j in i:
            print(str(j), sep=' ', end=' ',file=f)

```

```
print('',file=f)
```

Приклад. Дано текстовий файл який містить дані про матрицю записану в «природному» вигляді (кожен окремий рядок файлу має містити елементи відповідного рядка матриці записані через пропуск). Вичитати матрицю з файлу та знайти мінімальний елемент матриці.

```
fn=input("Введіть ім'я файлу: ")
m=[]
with open(fn,'r') as f:
    a=list(f)
    for i in a:
        b=list(map(int, i.split()))
        m.append(b)
min_col=list(map(min, m))
min_el=min(min_col)
print(min_el)
```

Вичитати матрицю з файлу можна і наступними операторами.

```
m=[]
with open('4.txt','r') as f:
    m=[list(map(int, i.split())) for i in list(f)]
```

Список використаних джерел

1. Python 3.8.0 documentation. URL: <https://docs.python.org/3/> (дата звернення: 12.11.2019).
2. Васильев А.Н. Python на примерах. Практический курс по программированию. 3-е издание. Москва: Наука и Техника, 2019. 432с.
3. Кольцов Д.М. Python. Создаем программы и игры. Москва: Наука и Техника, 2017. 400с.
4. Майкл МакГрат. Программирование на Python для начинающих: [перевод с англ. Райтман М.А.]. Москва: Эксмо, 2015. 192 с.
5. Матвійчук С.В., Жуковський С.С. Практикум програмування Python / C++ на e-olymp.com (збірник задач з рекомендаціями до їх розв'язання). Житомир: Вид-во ЖДУ ім. І. Франка, 2019. 232 с.
6. Програмування на мові Python (3.х). Початковий курс. URL: <https://sites.google.com/site/pythonukr/> (дата звернення: 12.11.2019).
7. Рамський Ю.С., Цибко Г.Ю. Основи програмування (мова Паскаль): навчальний посібник. Київ, 2003. 140 с.
8. Роберт СеджвикКевин УэйнРоберт Дондеро . Программирование на языке Python. Учебный курс: [перевод с англ. Коваленко В.А.]. Москва: Эксмо, 2017. 736 с.
9. Томашевский П. Привет, Python! Моя первая книга по программированию. Москва: Наука и Техника, 2018. 256с.
10. Федоров Д.Ю. Основы программирования на примере языка Python : учеб. пособие. СПб.: Наука и Техника, 2016. 176 с.
11. Эл Свейгарт. Учим Python, делая крутые игры: [перевод с англ. Райтман М.А.]. Москва: Эксмо, 2018. 416 с.

Додаток 1. Імпортування модулів та їх атрибутів

При написанні програм досить часто виникає необхідність в імпортуванні (підключенні) до власної програми модулів (бібліотек) для отримання доступу до їх атрибутів (елементів). Модулем у мові Python є файл, що містить змінні, підпрограмами, класи та ін. Варто зауважити, що збережена програма на мові Python також є модулем. Окрім того, модуль може бути написаний й іншими мовами, наприклад, C або C++.

Інструкція `import`

Імпортувати модуль можна за допомогою оператора `import`, після якого вказується назва модуля, що буде імпортовано (`import назва_модуля`). Наприклад, імпортування модуля `math`:

```
>>> import math
```

Після імпортування модуль стане доступним в поточній відкритій програмі, в якій його було імпортовано. Назва ж цього модуля стане змінною, через яку можна отримати доступ до атрибутів модуля (`назва_модуля.атрибут`).

Наприклад, звернення до константи `pi`, розташованої в модулі `math`:

```
>>> import math
>>> math.pi
3.141592653589793
```

Загалом, однією інструкцією може бути імпортовано декілька модулів.

```
>>> import math, random
```

Проте такий варіант не є рекомендованим, так як це знижує читабельність коду.

Під час імпортування модуля шляхи його місцезнаходження перевіряються за змінною `sys.path`. В ці шляхи також включена поточна директорія (тобто власний модуль можна розмістити в каталозі з основною програмою), а також директорія, в якій встановлено Python. Змінну `sys.path` можна змінювати вручну, що дозволяє розмістити модуль в будь-яке зручне для вас місце.

Варто відзначити, що якщо не вдасться знайти модуль для імпортування, то виникне виняток `ModuleNotFoundError`.

```
>>> import myunit
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    import myunit
ModuleNotFoundError: No module named 'myunit'
```

Використання псевдонімів

Якщо назва модуля занадто довга, або вона вам не подобається з інших причин, то замість неї може бути використаний псевдонім. Для вказання псевдоніму модуля передбачене ключове слово `as`:

```
import назва_модуля as псевдонім.
```

Наприклад, для скорочення імені модуля `math` до псевдоніму `m` необхідно записати:

```
>>> import math as m
```

В такому випадку для звернення до константи `pi` можна записати `m.pi`:

```
>>> m.pi
3.141592653589793
```

Інструкція `from ... import`

Досить часто виникає необхідність у використанні лише деяких атрибутів модуля. В такому випадку немає сенсу підключати весь модуль, а можна підключити лише ті атрибути, які є необхідними. Підключити певні атрибути модуля можна за допомогою інструкції `from ... import`, яка має кілька форматів:

```
from назва_модуля import
    атрибут1 [as псевдонім1 ],
    [атрибут2 [as псевдонім2 ] ...]
from назва_модуля import *
```

Перший формат дозволяє підключити з модуля тільки зазначені вами атрибути. Для довгих імен атрибутів також можна призначити псевдонім, вказавши його після ключового слова `as`.

```
>>> from math import pi, sqrt as sq
>>> pi
3.141592653589793
>>> sq(4)
```

2.0

Другий формат інструкції (`from назва_модуля import *`) дозволяє підключити атрибути з модуля. Для прикладу імпортуємо всі атрибути з модуля `math`:

```
>>> from math import *
>>> pi
3.141592653589793
>>> sqrt(4)
2.0
```

Слід зауважити, що не всі атрибути будуть підключені. Якщо в модулі визначена змінна `__all__` (список атрибутів, які можуть бути підключені), то будуть підключені тільки атрибути з цього списку. Якщо змінна `__all__` не визначена, то будуть підключені всі атрибути, які не починаються з нижнього підкреслення. Крім того, необхідно враховувати, що імпортування всіх атрибутів з модуля може порушити простір імен головної програми, так як змінні, що мають однакові імена, будуть перезаписані.

Додаток 2. Модуль `random`

Випадкові числа знаходять безліч застосувань в науці і програмуванні.

Модуль `random` містить функції для генерації випадкових чисел, букв або випадкового набору елементів послідовності. Для налагодження генератора випадкових чисел використовується системний час, тому при його ініціалізації кожен раз буде генеруватися нова послідовність.

Отримання випадкових чисел засноване на спеціальних алгоритмах, тобто вони не є дійсно випадковими, однак їх такими можна вважати. Інколи такі числа називають псевдовипадкові.

Для роботи з цим модулем його попередньо потрібно імпортувати, тобто вказати команду `import random`.

Функції генерації

`random.random()`. Повертає псевдовипадкове дійсне число з діапазону $[0.0, 1.0)$.

`random.randrange(stop)`. Повертає псевдовипадкове ціле число від 0 до $stop-1$.

`random.randrange(start, stop[, step])`. Повертає псевдовипадкове ціле число від 0 до $stop-1$ з кроком `step`. За замовчуванням параметр `step=1`.

```
>>> random.randrange(10)
5
>>> random.randrange(3, 10)
6
>>> random.randrange(3, 10, 3)
9
```

`random.uniform(A, B)`. Повертає псевдовипадкове дійсне число, що належить відрізьку $[A, B]$ або $[B, A]$.

`random.randint(A, B)`. Повертає псевдовипадкове ціле число, що належить відрізьку $[A, B]$.

`random.choice(sequence)`. Повертає випадковий елемент з послідовності `sequence` (рядка, списку, кортежу, тощо).

```
>>> random.choice('рядок')
```

```
'я'  
>>> random.choice(['a','b','c'])  
'b'
```

random.shuffle(sequence, [rand]). Перемішує та повертає елементи послідовності `sequence` (рядка, списку, кортежу, тощо). Оскільки змінюється сама послідовність, то функція не застосовна для незмінних об'єктів. Необов'язковий аргумент `rand` є функцією без аргументів, що повертає випадкове значення з діапазону `[0.0, 1.0)`, за замовчуванням використовуватиметься `random.random`.

```
>>> arr=[1,2,3,4,5,6,7,8,9,0]  
>>> random.shuffle(arr)  
>>> arr  
[9, 2, 6, 3, 1, 0, 7, 8, 5, 4]
```

random.sample(population, k). Повертає список з псевдовипадкових `k` елементів послідовності `population` (рядка, списку, кортежу, тощо). `k` не може бути більше за довжину послідовності `population`.

```
>>> random.sample('рядок', 2)  
['р', 'д']  
>>> arr=[1,2,3,4,5,6,7,8,9,0]  
>>> random.sample(arr,3)  
[7, 8, 3]  
>>> random.sample(range(300),5)  
[69, 161, 211, 23, 208]
```

random.getrandbits(N). Повертає псевдовипадкове ціле число, бітовий запис якого не перевищує `N` біт.

Функції налаштування генератора

random.seed(X=None, version=2). Налаштування стану генератора випадкових чисел на нову послідовність генерації. Для однакового значення параметра `X` генеруються однакові послідовності. Як значення параметра `X` за замовчуванням використовується системний час.

```
random.seed(20)  
random.random()
```

```
0.9056396761745207
random.random()
0.6862541570267026
random.seed(20)
random.random()
0.9056396761745207
random.random()
0.7665092563626442
```

random.getstate(). Повертає об'єкт, що фіксує поточний внутрішній стан генератора. Цей об'єкт може бути переданий в `setstate()` для відновлення стану.

random.setstate(state). Відновлює внутрішній стан генератора. Параметр `state` повинен бути отриманий функцією `getstate()`.

Імовірнісні розподіли

random.triangular(low, high, mode) – трикутний розподіл. Повертає псевдовипадкове дійсне число з відрізка $[low, high]$. Параметр `mode` задає режим розподілу $low \leq mode \leq high$.

http://en.wikipedia.org/wiki/Triangular_distribution

random.betavariate(alpha, beta) – бета-розподіл. Повертає псевдовипадкове дійсне число з відрізка $[0.0, 1.0]$. Параметри `alpha` і `beta` повинні бути більші нуля.

random.expovariate(lambd) – експоненціальний розподіл. Параметр `lambd` повинен бути відмінним від нуля і рівним $1/(\text{бажане середнє значення})$. Повертає псевдовипадкове число з інтервалу $(0, +\infty)$, якщо $lambd > 0$ або з інтервалу $(-\infty, 0)$, якщо $lambd < 0$.

random.gammavariate(alpha, beta) – гамма-розподіл. Параметри `alpha` і `beta` повинні бути більші нуля.

random.gauss(mu, sigma) – розподіл Гаусса. Параметр `mu` задає середнє значення, `sigma` – стандартне відхилення. Дещо швидше, ніж функція `random.normalvariate()`.

random.lognormvariate(mu, sigma) – логарифм нормального розподілу. Якщо взяти натуральний логарифм цього розподілу, то ви

отримаєте нормальний розподіл із середньою сігмою μ і стандартним відхиленням σ . μ може мати будь-яке значення, а σ повинна бути більше нуля.

`random.normalvariate(mu, sigma)` – нормальний розподіл. Параметр μ задає середнє значення, σ – стандартне відхилення.

`random.vonmisesvariate(mu, kappa)` – круговий розподіл. μ – середній кут (в радіанах від 0 до 2π), κ – параметр концентрації ($\kappa \geq 0$). Якщо κ дорівнює нулю, це розподіл зводиться до випадкового кутку в діапазоні від 0 до 2π .

`random.paretovariate(alpha)` – розподіл Парето. **`alpha`** – параметр форми.

`random.weibullvariate(alpha, beta)` – розподіл Вейбулла. α – параметр масштабу, β – параметр форми.

Приклад. Генерація випадкового пароля

Надійний пароль має містити мінімум 8 символів, серед яких можуть бути цифри, рядкові і прописні літери. Згенерувати такий пароль можна наступним чином:

```
import random
str1 = '123456789'
str2 = 'qwertyuiopasdfghjklzxcvbnm'
str3 = str2.upper()
str4 = str1+str2+str3
ls = list(str4)
random.shuffle(ls)
psw = ''.join([random.choice(ls) for x in range(8)])
print(psw)
1t9G4YPs
```

Цю ж програму можна записати так:

```
import random
print(''.join([random.choice(list('123456789qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) for x in range(8)]))
```

Додаток 3. Форматування рядків

Іноколи виникають ситуації, коли за програмою необхідно вивести повідомлення, що буде містити як текстові літерали, так і значення змінних. В такому випадку постає задача форматування рядка за певним шаблоном. Таке форматування можна виконати за допомогою оператора `%` або методу `format()`. Розглянемо застосування саме методу `format()`.

Метод `format`

Метод `format()` є методом рядка, тому він застосовується до рядка: `str.format(*args, **kwargs)`.

Рядок, для якого викликається цей метод, може містити явний текст та поля заміни, замість яких в подальшому за методом `format()` будуть підставлені конкретні значення, взяті з аргументів цього методу.

Полям заміни є пара фігурних дужок «`{}`», між якими можуть розміщуватися параметри налагодження заміни. Якщо потрібно включити символ фігурної дужки в явному тексті, його необхідно подвоїти: «`{{}}`» або «`}}`».

Підстановка значень

Найпростішим варіантом є підстановка лише одного значення.

```
>>>'Hello, {}'.format('World')
'Hello, World!'
```

Тобто в рядку `'Hello, {}'` міститься явний текст «Hello, » та єдине поле заміни, замість якого за методом `format()` буде підставлений аргумент `'World'`.

За необхідності сформувати рядок, в який будуть підставлятися декілька значень, необхідно в рядку розмістити відповідну кількість полів заміни, а в методі `format()` вказати необхідну кількість аргументів. Вказані аргументи будуть підставлені в поля заміни в порядку їх слідування. Відповідно кількість полів заміни та кількість аргументів має бути однаковою.

```
>>>'Точка ({}; {})'.format(3, 5)
'Точка (3; 5)'
```

Значення аргументів можуть бути отримані зі списку чи кортежу (використовуючи оператор `*`).

```

>>> d=[3,5]
>>> 'Точка ({}; {})'.format(*d)
'Точка (3; 5)'
>>> d=(3,5)
>>> 'Точка ({}; {})'.format(*d)
'Точка (3; 5)'
>>> 'Точка ({d[0]}; {d[1]})'.format(d=[3,5])
'Точка (3; 5)'
>>> 'Точка ({d[0]}; {d[1]})'.format(d=(3,5))
'Точка (3; 5)'

```

Для можливої зміни порядку підставлення аргументів в поля заміни чи уточнення, який саме аргумент буде підставлятися замість якого поля заміни, можна в поле заміни додати порядковий номер аргументу. Нумерація аргументів починається з 0.

```

>>> 'x1={0}, x2={1}, x3={2}'.format(1, 3, 5)
'x1=1, x2=3, x3=5'
>>> 'x1={2}, x2={1}, x3={0}'.format(1, 3, 5)
'x1=5, x2=3, x3=1'
>>> 'x1={0}, x2={1}, x3={0}'.format(1, 3)
'x1=1, x2=3, x3=1'
>>> 's1={0}, s2={1}'.format(*'13')
's1=1, s2=3'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'

```

Якщо використання нумерації аргументів недостатньо, то можна іменувати аргументи, або отримувати іменовані аргументи зі словника (використовуючи оператор **).

```

>>> 'Точка ({X}; {Y})'.format(X=3, Y=5)
'Точка (3; 5)'
>>> coord={'X':3, 'Y':5}
>>> 'Точка ({X}; {Y})'.format(**coord)
'Точка (3; 5)'

```

Перетворення аргументів

При форматуванні рядка аргументи методу `format()` перед підстановкою в поля заміни перетворюються за замовчуванням функцією `str()`. Проте за необхідності можна вказати інший метод перетворення. Для цього необхідно в поле заміни додати знак оклику «!» та вказати односимвольний код методу перекодування:

- a – перекодування функцією `str()`;
- r – перекодування функцією `repr()`;
- s – перекодування функцією `ascii()`.

Зрозуміло, таке перетворення доцільне тільки для текстових аргументів.

```
>>> '{0!s}{1!r}{0!a}'.format('abra', 'cad')
'abra'cad'abra'
>>> '{0!s}{1!r}{0!a}'.format('абра', 'кад')
'абра'кад'\u0430\u0431\u0440\u0430'\u0431\u0440\u0430'
>>> '{0}{1}{0}'.format('абра', 'кад')
'абракадабра'
```

Детально про функції `str()`, `repr()` та `ascii()` можна прочитати в розділі роботи з рядковими величинами.

Специфікація

Поле заміни також може містити специфікацію, за якою буде уточнюватися, як значення аргументу має бути представлено після підстановки до рядка. Специфікація вказується після двокрапки «:» та в загальному вигляді має формат:

```
[[fill]align][sign][#][0][width][grouping][.precision][type]
```

де: `fill` – заповнення (довільний символ), `align` – вирівнювання ("`<`" | "`>`" | "`=`" | "`^`"), `sign` – знак числа ("`+`" | "`-`" | " "), `#` – альтернативна форма подання, `0` – нуль підкладка, `width` – ширина поля (додатне ціле число), `grouping` – роздільник груп розрядів ("`_`" | "`,`"), `precision` – точність (додатне ціле число), `type` – тип аргументу ("`b`" | "`c`" | "`d`" | "`e`" | "`E`" | "`f`" | "`F`" | "`g`" | "`G`" | "`n`" | "`o`" | "`s`" | "`x`" | "`X`" | "`%`").

Встановлення ширини поля

При форматуванні рядка ширина поля (кількість символів поля) визначається значенням аргументу (кількістю символів необхідною для відображення аргументу після перетворення). За необхідності можна вказати ширину поля, тобто створити внутрішні відступи в полі заміни. Ширина поля вказується в символах, тобто є цілим числом. Встановлення ширини поля дозволяє візуально організувати велику кількість даних.

Для того, щоб вказати ширину поля, в специфікації поля заміни необхідно вказати ціле число (в фігурних дужках додати двокрапку «>» та вказати ширину поля).

```
>>> 'Точка ( {:5}; {:6} )'.format(-3, 5)
'Точка (   -3;      5 )'
>>> 'Точка ( {:5}; {:6} )'.format(-3, 5)
'Точка (   -3;      5 )'
>>> 'Точка ( {:5}; {:6} )'.format('X', 'Y')
'Точка (X      ; Y      )'
```

Вирівнювання значень

Як видно з наведених вище прикладів, за замовчуванням числа вирівнюються за правою межею, а рядкові значення – за лівою. За необхідності можна змінити вирівнювання, вказавши односимвольний код вирівнювання:

< – вирівнювання за правою межею;

> – вирівнювання за лівою межею;

^ – вирівнювання за центром;

= – використовується лише для числових даних, знак виводиться крайнім лівим символом, а саме число вирівнюється за правою межею (утворюється відступ між знаком та числом).

```
>>> 'Точка ( {:<5}; {:<6} )'.format(-3, 5)
'Точка (-3    ; 5    )'
>>> 'Точка ( {:^5}; {:^6} )'.format('X', 'Y')
'Точка (  X   ;   Y  )'
>>> 'Точка ( {:=5}; {:=6} )'.format(-3, 5)
'Точка (-  3;      5 )'
```


Зрозуміло, що вирівнювання застосовується лише при встановленні ширини поля.

Заповнення відступів

За замовчуванням відступи організуються додаванням пропусків (символ « »). За необхідності можна вказати символ, який буде використовуватися, замість пропусків для заповнення відступів. Для цього на початку специфікації (після двокрапки) вказується необхідний символ (окрім символів фігурних дужок «{» та «}»).

```
>>> 'Точка ( {:*<5}; {:*<6} )'.format(-3, 5)
'Точка (-3***; 5*****)'
>>> 'Точка ( {:_^5}; {:_^6} )'.format('X', 'Y')
'Точка ( __X__ ; __Y__ )'
>>> 'Точка ( {:0=5}; {:0=6} )'.format(-3, 5)
'Точка (-0003; 000005)'
```

Зрозуміло, що використання символу заповнення, так само як і вирівнювання, застосовується лише при встановленні ширини поля. Проте також не допускається використання символу заповнення без явного вказання вирівнювання.

Якщо ніякого явного вирівнювання для числових типів не задано, перед полем ширини передбачено нульову підкладку (символ «0»). Це еквівалентно знаку заповнення '0' з типом вирівнювання '='.

```
>>> 'Точка ( {:05}; {:06} )'.format(-3, 5)
'Точка (-0003; 000005)'
```

Відображення знаку числа

Опція знаку числа використовується лише для чисел і може мати такі значення:

- - знак відображається лише для від'ємних чисел (значення за замовчуванням);

+ - знак відображається як для від'ємних, так і для додатних чисел (відповідно символи «-» та «+»);

пропуск - для від'ємних чисел використовується символ «-», для додатних - символ пропуску « ».

```
>>> 'Точка ( {:+}; {:+} )'.format(-3, 5)
```

```
'Точка (-3; +5) '  
>>> 'Точка ([:-]; {:-})'.format(-3, 5)  
'Точка (-3; 5) '  
>>> 'Точка ([: ]; {: })'.format(-3, 5)  
'Точка (-3; 5) '
```

Відокремлення груп розрядів

За замовчуванням при виведенні чисел всі цифри числа записуються без розривів (групи розрядів не відокремлені). За необхідності можна вказати символ, який буде використовуватися для відокремлення груп розрядів (тисяч). Таким символом може бути кома «,» або нижнє підкреслення «_».

```
>>> 'Точка ({}; {})'.format(-30000, 50000)  
'Точка (-30000; 50000) '  
>>> 'Точка ({:},); {:_})'.format(-30000, 50000)  
'Точка (-30,000; 50_000) '
```

Типи аргументів

Поле заміни також може містити вказівку на тип даних, що будуть виводитися. Для цього необхідно вказати односимвольний код типу даних. Виділяють рядкові, цілі та дійсні типи.

Рядкові типи:

s – рядковий формат. Тип за замовчуванням для рядкових аргументів.

None – те ж саме, що і «s»

Цілі типи:

b – бінарний (двійковий) формат. Виведення числа за основою 2.

c – символ. Перетворює ціле число у відповідний символ юнікоду перед виведенням.

d – десяткове ціле число. Виведення числа за основою 10.

o – вісімковий формат. Виведення числа за основою 8.

x – шістнадцятковий формат. Виведення числа за основою 16, використовуючи рядкові літери для цифр більше 9.

X – те ж саме, що і «x», лише використовуються літери в верхньому регістрі.

n – число. Теж саме, що і «d», лише використовуються поточні регіональні налаштування для вставки символів роздільника цифр.

None – те ж саме, що і «d»

Дійсні типи:

e – експонентна (наукова) форма (експонента «e» в нижньому регістрі). Точність за замовчуванням рівна 6 знакам після коми.

E – те ж саме, що і «e», лише експонента «E» в верхньому регістрі.

f – фіксований формат. Виводить число з фіксованою точкою. Точність за замовчуванням рівна 6 знакам після коми.

F – те ж саме, що і «f», лише перетворює nan та inf в верхній регістр, відповідно NAN та INF.

g – загальний формат. Числа менші 10^{-4} виводяться в експонентній формі, інші числа виводяться в фіксованому форматі (не більше 9 знаків після коми).

G – те ж саме, що і «g», лише при виведення експоненти «E» вона записується в верхньому регістрі, nan та inf також в верхньому регістрі (NAN та INF).

n – число. Те ж саме, що і «g», лише використовуються поточні регіональні налаштування для вставки символів роздільника цифр.

% – відсоток. Добуток числа на 100, відображеного в фіксованому форматі «f».

None – Те саме, що і «g», за винятком того, що значення з фіксованою точкою має принаймні одну цифру перед десятковою крапкою. Точність за замовчуванням настільки висока, наскільки потрібно для відображення певного значення. Загальний ефект полягає в тому, щоб відповідати виводу str(), зміненому іншими модифікаторами формату.

```
>>> 'int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}'  
      .format(42)  
'int: 42; hex: 2a; oct: 52; bin: 101010'  
>>> 'Точка ({:f}, {:f})'.format(math.pi/10000,  
      math.pi/100000)  
'Точка (0.000314, 0.000031)'  
>>> 'Точка ({:g}, {:g})'.format(math.pi/10000,  
      math.pi/100000)  
'Точка (0.000314159, 3.14159e-05)'
```

```
>>> 'Точка ( {:.%}, {:.%})'.format(math.pi/10000,
                                     math.pi/100000)
'Точка (0.031416%, 0.003142%)'
```

Точності.

Точність - це число, яке вказує, скільки цифр слід відображати після десяткової коми для значень, відформатованих у фіксованому форматі (за допомогою "f" та "F"), або скільки цифр слід відображати до і після десяткової коми (загальна кількість цифр без врахування знаку числа та десяткової коми) для значень, відформатованих в загальному форматі (за допомогою "g" або "G"). Для рядкових даних точність вказує максимальний розмір поля - іншими словами, скільки символів буде використано з вмісту поля.

```
>>> 'Точка ( {:.f}, {:.5f})'.format(math.pi/100000,
                                     math.pi/100000)
'Точка (0.000031, 0.00003) '
>>> 'Точка ( {:.f}, {:.5g})'.format(100.123456,
                                     100.123456)
'Точка (100.123456, 100.12) '
>>> ' {:.2}'.format('Hello')
'He'
```

Альтернативна форма перетворення

Опцією «#» вказується необхідність використання для перетворення так званої «альтернативної форми». Ця опція використовується лише при заданні цілих (int), дійсних (float), комплексних (complex) та decimal чисел. Альтернативна форма визначається по-різному для різних типів. Для цілих чисел, коли використовується двійкове, вісімкове чи шістнадцяткове виведення, за цією опцією додається відповідний префікс «0b», «0o» або «0x» до вихідного значення. Для інших чисел альтернативна форма приводить до того, що результат перетворення завжди містить знак десяткової коми, навіть якщо після неї немає жодної цифри. Окрім того, для загального формату («g» або «G») кінцеві нулі не прибираються з результату.

```
>>> 'hex: {0:x}; oct: {0:o}; bin: {0:b}' .format(42)
'hex: 2a; oct: 52; bin: 101010'
```

```

>>> 'hex: {0:#x}; oct: {0:#o}; bin: {0:#b}'
      .format(42)
'hex: 0x2a; oct: 0o52; bin: 0b101010'
>>> 'float_f: {0:#f}; float_g: {0:#g}' .format(42)
'float_f: 42.000000; float_g: 42.0000'
>>> 'float_f: {0:#f}; float_g: {0:#g}'
      .format(123456)
'float_f: 123456.000000; float_g: 123456.'

```

Задання полів специфікаторів

Поле заміни також може містити вкладені поля заміни, але не більше одного вкладення. Ці вкладені поля заміни можуть використовуватися для динамічного вказування параметрів форматування поля. Таке використання дозволяє динамічно вказувати форматування значень.

```

>>> 'Точка ({:_^5}; {:_^6})'.format('X', 'Y')
'Точка (__X__; __Y__)'
>>> 'Точка ({:{fill}{align}5};
      {:{fill}{align}6})'.format('X',
      'Y',fill='_',align='^')
'Точка (__X__; __Y__)'
>>> for i in range(1,5):
      '{:{wd}}'.format(1, 1, wd=i)
'1'
' 1'
'  1'
'   1'
>>> for i in range(1,7):
      'Pi={:.{prec}f}'.format(math.pi, prec=i)
'Pi=3.1'
'Pi=3.14'
'Pi=3.142'
'Pi=3.1416'
'Pi=3.14159'
'Pi=3.141593'

```

Додаток 4. Дерево класів Exception

Імена стандартних винятків являють собою вбудовані ідентифікатори (не зарезервовані ключові слова). Розглянемо ієрархію винятків, вбудованих в мову Python, хоча іноді вам можуть зустрітися і інші, так як програмісти можуть створювати власні винятки. Наведений список актуальний для python 3.7, в більш ранніх версіях є незначні зміни.

- **BaseException** – базовий клас винятків, від якого беруть початок всі інші.
 - **SystemExit** - породжується функцією `sys.exit` при виході з програми.
 - **KeyboardInterrupt** - породжується при перериванні програми користувачем (зазвичай комбінацію кнопок `Ctrl + C`).
 - **GeneratorExit** - породжується при виклику методу `close` об'єкта `generator`.
 - **Exception** – базовий клас для всіх вбудованих несистемних винятків. Всі винятки, визначені користувачем, також повинні бути нащадками цього класу.
 - **StopIteration** - породжується вбудованою функцією `next`, якщо в ітераторі більше немає елементів.
 - **StopAsyncIteration** - породжується вбудованою функцією `anext`, якщо в асинхронному ітераторі більше немає елементів.
 - **ArithmeticError** – виняток арифметичних помилок.
 - **FloatingPointError** - невдале виконання операції з дійсними числами.
 - **OverflowError** - результат арифметичної операції занадто великий для подання.
 - **ZeroDivisionError** - ділення на нуль.
 - **AssertionError** - вираз у функції `assert` хибний.
 - **AttributeError** – об'єкт не має даного атрибута (значення або методу).
 - **BufferError** - операція, пов'язана з буфером, не може бути виконана.
 - **EOFError** - функція натрапила на кінець файлу і не змогла прочитати те, що вказувалося.

- **ImportError** - не вдалося імпортування модуля або його атрибута.
 - **ModuleNotFoundError** - не вдалося знайти модуль при імпортуванні.
- **LookupError** - некоректний індекс або ключ.
 - **IndexError** - індекс не входить в діапазон елементів.
 - **KeyError** - неіснуючий ключ (в словнику, множині і т.д.).
- **MemoryError** - недостатньо пам'яті.
- **NameError** - не знайдено змінної з таким ім'ям.
 - **UnboundLocalError** - зроблено посилання на локальну змінну у функції, але змінна раніше не визначена.
- **OSError** - помилка, пов'язана з системою.
 - **BlockingIOError** – операція блокується на об'єкті.
 - **ChildProcessError** - невдача при операції з дочірнім процесом.
 - **ConnectionError** - базовий клас для винятків, пов'язаних з підключеннями, а саме: **BrokenPipeError**, **ConnectionAbortedError**, **ConnectionRefusedError**, **ConnectionResetError**.
 - **FileExistsError** - спроба створення файлу або каталогу, який вже існує.
 - **FileNotFoundError** - файл або каталог не існує.
 - **InterruptedError** - системний виклик перерваний вхідним сигналом.
 - **IsADirectoryError** - очікувався файл, але це каталог.
 - **NotADirectoryError** - очікувався каталог, але це файл.
 - **PermissionError** - не вистачає прав доступу.
 - **ProcessLookupError** - вказаного процесу не існує.
 - **TimeoutError** - закінчився час очікування.
- **ReferenceError** - спроба доступу до атрибуту зі слабким посиланням.
- **RuntimeError** - виникає, коли виняток не потрапляє ні під одну з інших категорій.

- **NotImplementedError** - абстрактні методи класу вимагають перевизначення в дочірніх класах.
 - **RecursionError** – інтерпретатор виявляє, що максимальна глибина рекурсії перевищена.
- **SyntaxError** - синтаксична помилка.
 - **IndentationError** - неправильні відступи.
 - **TabError** - змішування в відступах табуляторів і пропусків.
- **SystemError** - внутрішня помилка.
- **TypeError** - операція застосована до об'єкта невідповідного типу.
- **ValueError** - функція отримує аргумент правильного типу, але некоректного значення.
 - **UnicodeError** - помилка, пов'язана з кодуванням / розкодуванням unicode в рядках.
 - **UnicodeEncodeError** - кодування unicode.
 - **UnicodeDecodeError** - декодування unicode.
 - **UnicodeTranslateError** - перекладення unicode.
- **Warning** – базовий клас для винятків, пов'язаних з попередженнями.
 - **UserWarning** - клас для попереджень, що генеруються кодом користувача
 - **DeprecationWarning** - клас для попереджень про застарілі функції, коли ці попередження призначені для інших розробників Python.
 - **PendingDeprecationWarning** - клас для попереджень про функції, які в майбутньому будуть застарілими.
 - **RuntimeWarning** - клас для попереджень про сумнівну поведінку під час виконання.
 - **SyntaxWarning** - клас для попереджень про сумнівний синтаксис.
 - **FutureWarning** - клас для попереджень про застарілі функції, коли ці попередження призначені для кінцевих

користувачів програм, написаних на Python.

- **ImportWarning** - клас для попереджень про можливі помилки в імпорті модулів.
- **UnicodeWarning** - клас для попереджень, пов'язаних з Unicode.
- **BytesWarning** - клас для попереджень, що відносяться до байтів і bytearray.
- **ResourceWarning** - клас для попереджень, пов'язаних з використанням ресурсів.

Навчальне видання

Основи програмування мовою Python

Костюченко Андрій Олександрович – кандидат педагогічних наук, старший викладач кафедри інформатики і обчислювальної техніки Національного університету «Чернігівський колегіум» імені Т.Г.Шевченка

Рецензенти:

Покришень Дмитро Анатолійович - кандидат педагогічних наук, доцент, завідувач кафедри природничо-математичних дисциплін та інформаційно-комунікаційних технологій в освіті Чернігівського обласного інституту післядипломної педагогічної освіти імені К.Д. Ушинського

Цибко Ганна Юхимівна - кандидат педагогічних наук, доцент кафедри інформатики і обчислювальної техніки Національного університету «Чернігівський колегіум» імені Т.Г.Шевченка

Підписано до друку 05.02.2020р. Формат 60x84/16
Папір офсетний. Гарнітура Таймс. Друк на Ризографі.

Ум.друк.арк. 11,25.

Тираж 100 прим. Зам. № 231.

Віддруковано в авторській редакції

ФОП Баликіна С.М.
м. Чернігів, пров. 1-го Травня, 2а
Тел.: (068)152-87-63
0462print@ukr.net