МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

М. А. Новотарський

ОСНОВИ НАУКИ ПРО ДАНІ

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за освітньою програмою
«Інженерія програмного забезпечення комп'ютерних систем»,
за спеціальністю
121 «Інженерія програмного забезпечення»*

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензенти: *Чемерис О. А.*, д. т. н., с. н. с., заступник директора з наукової роботи Інституту проблем моделювання в енергетиці ім. Г. Є. Пухова НАН України

Відповідальний редактор *Стіренко С. Г.*, д. т. н., проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського

(протокол № 1 від 02.09.2022 р.)

за поданням Вченої ради Факультету інформатики та обчислювальної техніки (протокол № 11 від 11.07.2022 р.)

Електронне мережеве навчальне видання

ОСНОВИ НАУКИ ПРО ДАНІ

Основи науки про дані [Електронний ресурс]: навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення» / М. А. Новотарський; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 5,5 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 294 с.

Навчальний посібник створений за матеріалами лекцій з курсу «Основи науки про дані». Він містить базові відомості про дані, їх категорії, види та шкали вимірювання. Коротко наведено основні положення алгоритмічної мови програмування Python, оскільки цю мову найчастіше застосовують для побудови алгоритмів аналізу даних. Розглянуто методи та функції бібліотеки Pandas, яка є популярною бібліотекою, що використовується для попередньої підготовки даних. Типові задачі аналізу даних представлені задачею логістичної регресії та задачею глибокого машинного навчання. Для практичної реалізації цих задач у вигляді комп'ютерних програм розглянуто фреймворк PyTorch, який має невисокий поріг входження при вирішенні різноманітних задач машинного навчання.

Укладач: *Новотарський Михайло Анатолійович*, д-р техн. наук, с.н.с.

ЗМІСТ

1. БАЗОВІ ВІДОМОСТІ ПРО ДАНІ

1.1. Поняття даних та одиниці вимірювання

Дані – це широке філософське поняття, яке у загальному випадку є продуктом діяльності людини. В кожній галузі науки дані характеризуються своєю специфікою. В сфері комп'ютерної інженерії ми можемо розглядати дані як інформацію, яка представлена у певній формі, що є зручною для передачі та зберігання. Для сучасних традиційних комп'ютерів дані – це інформація, яка представлена у бінарній формі (тобто у вигляді нулів або одиниць). Комп'ютери можуть представляти дані у вигляді, який ми називаємо відео, звук, текст і т.ін. Але у кожному з цих випадків ми матимемо дані, які представлені у вигляді послідовності нулів та одиниць. Певний фізичний об'єкт, який може знаходитися лише у одному з двох можливих станів, є об'єктом, який зберігає один біт інформації. Таким чином, біт – це найменша одиниця даних. В комп'ютерах прийнято представляти інформацію наборами по 8 біт. Такі набори даних називають байтами. Для зберігання більших обсягів даних прийнято використовувати похідні одиниці вимірювання, які наведено в таблиці 1.1

Таблиця 1.1.

Вимірювання обсягів даних

Одиниця вимірювання	**Значення**
Біт	1 біт
Байт	8 бітів
кілобайт	1024 байт
мегабайт	1024 кілобайт
гігабайт	1024 мегабайт
терабайт	1024 гігабайт
петабайт	1024 терабайт
ексабайт	1024 петабайт
зеттабайт	1024 ексабайт
йотабайт	1024 зеттабайт
бронтобайт	1024 зеттабайти

1.2. Категорії даних

Дані, які потребують певного набору методів для зберігання та обробки, утворюють категорію даних. В науці про дані використовують такі категорії даних:

1. Структуровані дані.
2. Неструктуровані дані.

Розглянемо детально кожну з цих категрій.

1.2.1. Структуровані дані

Розглянемо таблицю 1.2, яка є набором структурованих даних.

Таблиця 1.2.

Структуровані дані

Код співробітника	Вік	Посада	Заробітна плата
1	18	Директор	125 000
2	22	Головний інженер	100 000
3	30	Начальник цеху	70 000
4	32	Бухгалтер	50 0000
5	24	Майстер	45 000
6	25	Слюсар	20 000
7	32	Токар	18 0000
8	19	Токар	15 000
9	22	Пакувальник	12 000
10	40	Вантажник	10 000

Кожний рядок таблиці відповідає одному об’єкту, який індексується кодом співробітника. Цей код розміщено у першому стовпці таблиці. Всі решта полів кожного рідка називатимемо атрибутами об’єкта. У термінології, яка використовується у базах даних, кожен рядок називають записом. Отже, запис включає множину атрибутів, які описують об’єкт. Оскільки кожен об’єкт має свою характеристику, то атрибути об’єкта є змінними.

Змінна (`variable`) – це загальна для всіх записів характеристика атрибуту, значення якої може змінюватися.

Значення (`value`) змінної визначає кількісну оцінку атрибуту.

Отже, структуровані дані – це високоорганізована, фактична і точна інформація. Зазвичай структуровані дані представлені у формі букв і цифр, які розміщені в рядках таблиць. Структуровані дані зазвичай представлені в таблицях, файлах Excel, електронних таблицях Google Docs.

1.2.2. Неструктуровані дані

Неструктуровані дані не мають заданої структури і можуть мати різні формати представлення. Приклади неструктурованих даних: зображення, що представлені форматами *.jpg, *png, *.tif та ін., текстові файли з форматами *.txt, *.docx, *.odt та ін. та документи PDF. До неструктурованих даних також відносять відео (*.mp4) і аудіо (*.mp3) файли.

Неструктуровані дані становлять близько 80% накопичених на сьогодні даних. Такі дані зберігаються у так званих озерах даних, що містять дані у первинних форматах. Ці дані є джерелом цінної інформації, яка може істотно допомогти при веденні бізнесу та у наукових дослідженнях.

Існує велика кількість джерел отримання неструктурованих даних. Для фірм та організацій, які володіють веб-порталами, таке накопичення відбувається природним чином. У цьому випадку неструктуровані дані можуть включати архіви веб-порталів, результати спілкування користувачів у чатах та форумах, які відкриті на веб-порталах, та ін. Неструктуровані дані можуть формуватися також цілеспрямовано всіма бажаючими з використанням інструментів мережі Інтернет або будь-якими іншими джерелами інформації.

Джерелом неструктурованих текстових даних може бути, наприклад, форум порталу «Українська правда», титульна сторінка якого наведена на рис. 1.1.

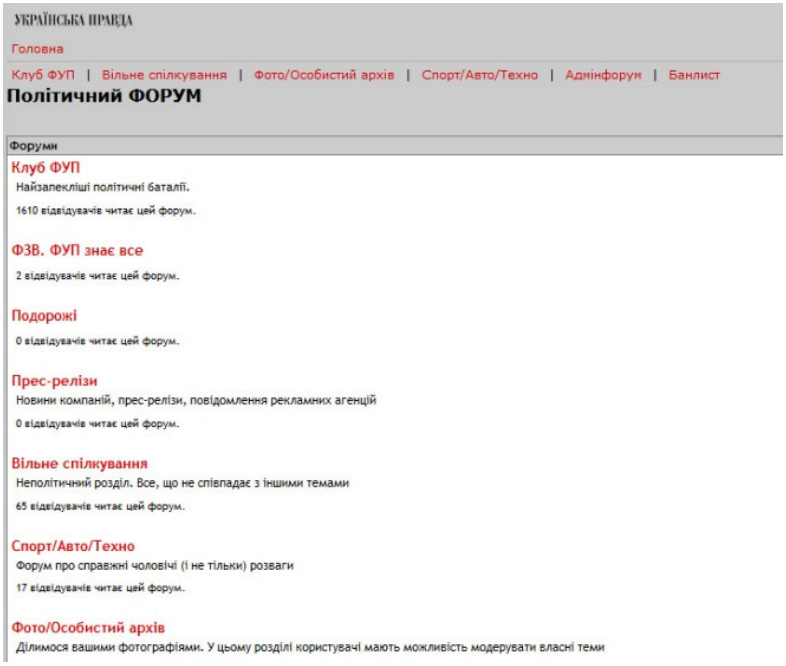


Рис. 1.1. Приклад джерела неструктурованих текстових даних

Неструктуровані дані у вигляді картинок ми можемо отримати, якщо задамо відповідний запит у пошуковій системі Google.

Приклад результату запиту «Неструктуровані дані» показаний на рис. 1.2.

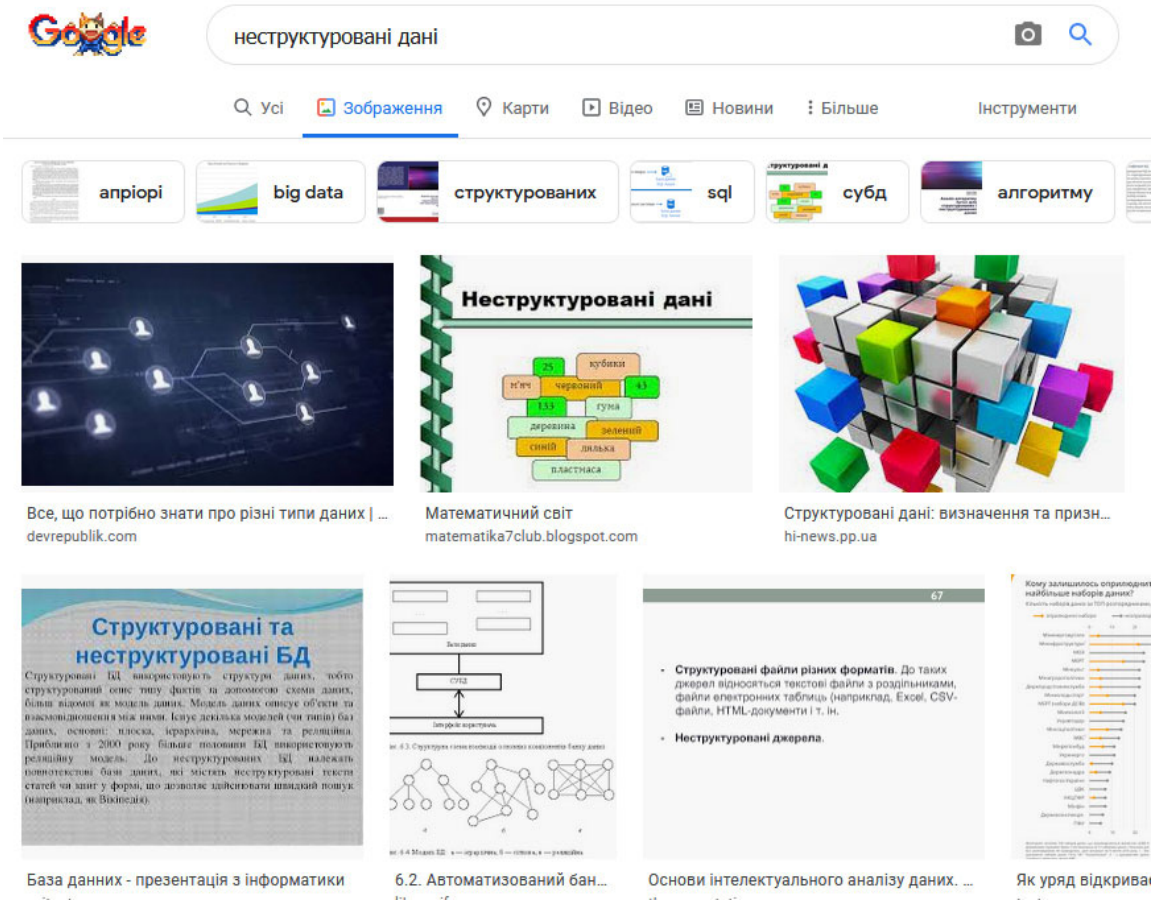


Рис. 1.2. Неструктуровані дані у вигляді зображень

1.3. Типи наборів даних

Розглянемо типи наборів даних, які будемо поділяти за джерелами їх походження та, відповідно, за сферами застосування. У цьому випадку дані поділяються за такими найбільш вживаними типами:

1. Дані природною мовою.
2. Дані в машинних кодах.
3. Дані, представлені графами.
4. Мультимедійні дані.
5. Потокові дані.

1.3.1. Дані природною мовою

Дані природною мовою відносяться до категорії неструктурованих даних. У цьому випадку маємо на увазі дані, які представлені у одному з текстових форматів. Обсяги цих даних, накопичених людством за всю історію, не є надзвичайно великими, у порівнянні з загальним обсягом даних, якщо розглядати їх представлення у вигляді кількості бітів. Але ці дані є найціннішим надбанням. Тому їх дослідження та обробка має велику актуальність. Крім тестових даних природною мовою існують також дані у звуковому та відео представленні.

У науці про дані виникла ціла область, яка має назву «Обробка природної мови» (NLP – Natural language processing). Обробка даних, які представлені природною мовою, пов’язана зі значними труднощами через такі особливості:

1. В основі конструкції природних мов лежить ідея компактної передачі сенсу сказаного або написаного. Це передача інформації у закодованому вигляді з великою складністю кодування.

2. В основі мови лежить символьна категоріальна система, яка є складною для комп’ютерної реалізації, оскільки потребує великої кількості ресурсів.

3. Існують додаткові канали передачі інформації, що можуть істотно змінити сенс звукової інформації. Тому розпізнавання звуку повинно супроводжуватися розпізнаванням образів, емоцій і т. ін.

Серед сучасних актуальних застосувань NLP є такі:

1. Пошук інформації в Інтернет (пошуковики Google, Yahoo, DuckDuckGo та ін.) дозволяє за певним набором слів побудувати відсортовану за релевантністю послідовність сторінок.

2. Таргетована реклама (Targeted advertising) у соціальних мережах.

Така реклама дозволяє пропонувати людям у соціальних мережах товари та послуги, якими вони з високою імовірністю можуть зацікавитися.

3. Комп'ютерний переклад текстів. Це надзвичайно актуальна обробки текстів, складність якої полягає у необхідності правильного перекладу сенсу сказаного з використанням правильних граматичних конструкцій цільової мови.

4. Використання доступної інформації природною мовою. Наприклад, можна вирішувати задачу ефективного маркетингу, базуючись на відгуках покупців та сучасних течіях у суспільстві.

5. Розпізнавання природної мови для автоматизації спілкування за допомогою чат-ботів.

6. Голосові помічники. Великий прорив у цьому напрямку досягнуто у сфері Інтернету речей. Як приклад можна навести систему Alexa.

1.3.2. Дані в машинних кодах

Це дані, які у загальному випадку представлені у вигляді послідовності одиниць та нулів. Машинний код для мікропроцесора має певні рівні структурування. Логічно завершений набір даних у машинному коді будемо називати програмою. Програма у машинному коді представлена послідовністю адресованих команд та даних, які зчитуються з використанням лічильника, що вказує поточну адресу.

На рис. 1.3 показано приклад вікна редактора, який дозволяє перегляд даних в машинних кодах.

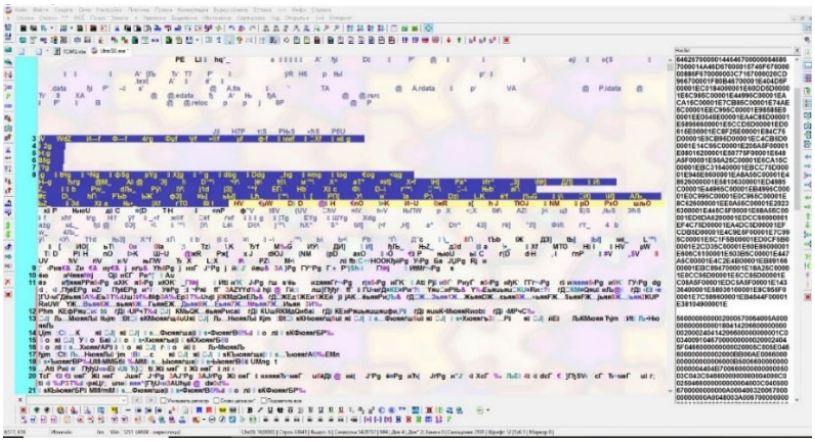


Рис.1.3. Відображення машинного коду

Таке представлення інформації є зручним для машини, але складним для розуміння людиною. Для полегшення побудови програм у машинних кодах застосовують алгоритмічні мови низького рівня або мови асемблера. Програма асемблера містить коди операцій, які представлені у вигляді мнемоніки та синтаксичних правил.

1.3.3. Дані, представлені графами

Графи можна розглядати як окремий тип набору даних, що включає реалізацію концепцій неорієнтованого і орієнтованого графів та орієнтованого двочасткового графа. Графи є абстрактним типом набору даних, тому можуть представляти різні структуровані дані для наглядного відображення характеру зв'язків між їх елементами.

Неорієнтований граф G задають кортежем $G=(V,E)$, де $V=\{v_1,v_2,...,v_n\}$ – множина вершин графа, $E=\{e_1,e_2,...,e_m\}$ – множина ребер. Кожне ребро представлене кортежем $(v_i,v_j)=(v_j,v_i)$.

Приклад неорієнтованого графа показаний на рис. 1.4.

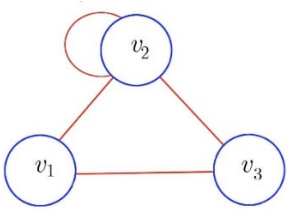


Рис. 1.4. Неорієнтований граф

Аналітичний запис такого графа: $G=(V,E)$, де

$V=(v_1,v_2,v_3), E=\{(v_1,v_2),(v_1,v_3),(v_2,v_3),(v_2,v_2)\}$, $E=E^{-1}$, $V\neq\varnothing$

Орієнтований граф G_d задають кортежем $G_d=(V,E)$, де $V=\{v_1,v_2,...,v_n\}$ – множина вершин вузлів, $E=\{e_1,e_2,...,e_m\}$ – множина дуг. Кожна дуга представлена кортежем $(v_i,v_j)\neq(v_j,v_i)$.

Приклад орієнтованого графа показаний на рис. 1.5.

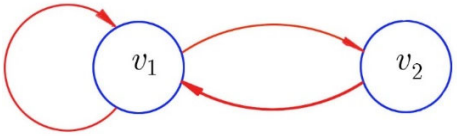


Рис. 1.5. Орієнтований граф

Аналітичний запис орграфа: $G_d=(V,E)$, де

$V=(v_1,v_2), E=\{(v_1,v_2),(v_2,v_1),(v_1,v_1)\}$, $E\neq E^{-1}$, $V\neq\varnothing$

Двочастковий граф Ф формально представляють також у вигляді кортежу. $\Phi=(P,T,F,M)$, P – множина позицій, T – множина переходів, $F=P\times T\cup T\times P$ множина зв'язків між позиціями та переходами, M – множина міток.

Приклад двочасткового графа показаний на рис 1.6.

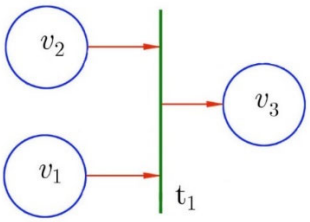


Рис. 1.6. Двочастковий граф

Аналітичний запис даного двочасткового графа: $\Phi=(P,T,F,M)$, де

$$P=\{v_1,v_2,v_3\},\ T=\{t_1\},\ F=\begin{pmatrix} & t_1 \\ v_1 & 1 \\ v_2 & 1 \\ v_3 & -1 \end{pmatrix},\ M=\varnothing$$

1.4. Мультимедійні дані

Аудіо – це дані про звукові коливання повітря, які записані на електронному носії інформації.

Аналогове аудіо – дані про звук, записані у аналоговому форматі (магнітна плівка, музичні пластинки).

Цифрове аудіо – дані про звук, записані у цифровому форматі (компакт диск, DVD – диск, флеш-пам'ять).

Для роботи зі звуковими даними використовують звукову карту, яка може входити до складу обладнання материнської плати або вставлятися в слот як додаткове обладнання. До складу звукової карти входить аналого-цифровий перетворювач для запису звуку та цифро-аналоговий перетворювач для його відтворення.

Відео. Відеодані представлені зображеннями, зафіксованими на цифрових носіях інформації.

Найпростіший вид роботи з відеоданими: перегляд фільмів та зображень.

Відеодані можуть бути статичними та динамічними.

Статичні відеодані: зображення тексту, рисунки, графіки, креслення, таблиці та ін. (формати jpg, pcx, tif, heic та ін.)

Динамічні відеодані: відеофільми, слайд-шоу, мультфільми (формати: AVI, MP4 та ін.)

Зображення можуть бути растровими або векторними. Растрові зображення формуються як множина пікселів з відповідними характеристиками кольору та яскравості. Векторні зображення формуються як сукупність геометричних фігур, тому векторні зображення можуть легко масштабуватися. Існують відповідні редактори для модифікації таких зображень.

1.4.1. Потокові дані

Типи наборів даних, які ми розглядали до цього часу, можуть бути збережені в базах даних. Тобто весь набір даних нам доступний в будь-який момент часу.

Кожен елемент потокових даних стає доступним користувачеві послідовно і в незалежний від користувача момент часу. Для того, щоб черговий елемент

потокових даних був використаний, його потрібно негайно обробити. У протилежному випадку цей елемент буде втрачено назавжди. Як правило, швидкість надходження потокових даних та їх загальний обсяг настільки великі, що такі дані або неможливо, або недоцільно попередньо зберігати з метою подальшої обробки.

Методи обробки таких даних становлять напрям, що має назву *аналіз потоків даних* і є складовою частиною науки про дані.

1.5. Шкали для вимірювання даних

Шкалою називатимем відображення якісних і кількісних характеристик даних на множину чисел або множину інших упорядкованих елементів (кольорів, символів алфавіту) .

Відповідно до «Міжнародної конвенції» створено Міжнародне бюро мір і ваг, основною метою якого є зберігання еталонів та точних шкал. Існує галузь науки, яку називають метрологією. Саме працівники цієї галузі займаються проблемами вимірювання даних.

Шкали поділяють на метричні та неметричні. Метричні шкали – це такі відображення характеристик даних, які можна кількісно визначити, та мають одиниці вимірювання. Неметричні шкали – це відображення, які не мають одиниць вимірювання.

Розглянемо основні типи шкал за класифікацією С.С. Стівенса.

1.5.1. Шкала найменувань або номінальна шкала

Це найпростіший неметричний тип шкал, який відображає якісні характеристики даних. Для номінальних шкал не використовують одиниці вимірювання. Хоча номінальні шкали можуть відображати елементи даних на множину чисел, у них відсутній нульовий елемент даних. Ці шкали також не допускають виконання арифметичних операцій. Для даних шкал існує відношення еквівалентності та подібності.

Приклади шкали найменувань: імена людей, назви факультетів КПІ, назви студентських груп, сімейний стан.

1.5.2. Шкала порядку або рангова шкала

Шкали порядку – це неметричний тип шкал, який відображає кількісні властивості даних. Для цього типу шкал характерним є відношення еквівалентності та упорядкування за величиною кількісної властивості. Найчастіше кількісну властивість визначають у балах, які є непорівнювальними з балами, що використовуються в іншій порядковій шкалі.

Дані, які вимірюються порядковими шкалами, називають порядковими даними. Величина порядкових даних визначається відповідно до прийнятої методики вимірювання. Для шкал порядку не використовують арифметичні операції. Ці дані не відповідають системі фізичних величин і тому не мають одиниць вимірювання у системі SI.

Приклади шкали порядку: рейтинговий бал студента за університетською шкалою, твердість матеріалу за шкалою Роквела, сила землетрусу за шкалою Ріхтера.

1.5.3. Шкала інтервалів (різниць)

Шкали порядку відносять до метричних типів шкал, які відображають кількісні властивості даних. Для шкал інтервалів характерними є відношення еквівалентності, відношення порядку та застосовні додавання та віднімання. У шкалах інтервалів можуть використовуватися одиниці вимірювання. Нульові елементи можуть бути наявні, якщо це допускається попередньою угодою. До наборів даних з шкалою інтервалів застосовними є операції математичної статистики: математичне сподівання, дисперсія, визначення моментів вищого порядку.

Приклади шкали інтервалів: температурні шкали за Цельсієм та Фаренгейтом, шкали розмірів.

1.5.4. Шкала відношень

Шкала відношень є шкалою метричного типу та відображає кількісні властивості даних. Для цієї шкали характерні відношення еквівалентності, порядку,

пропорційності проявів властивості. Всі шкали відношень мають природний абсолютний нуль та одиниці вимірювання.

Для даних цього типу доступні статистичні операції та арифметичні операції. Ці шкали є такими, що широко використовуються у фізиці та техніці. Для даних, відображених шкалою відношень, можна визначити, у скільки разів вони відрізняються, тобто знайти відношення між ними.

Приклади шкали відношень: шкала термодинамічних температур, шкала мас.

1.5.5. Абсолютна та додаткові шкали

Основна властивість абсолютної шкали – природна і однозначна наявність одиниць вимірювання. Шкала має природну нульову точку, у якій відсутня кількісна ознака за абсолютною шкалою.

Приклад абсолютної шкали: кількість студентів в аудиторії, кількість відмінників у групі.

Додатково використовують логарифмічні та біофізичні шкали. Для отримання логарифмічної шкали необхідно взяти логарифм від довільної метричної шкали. Можна використовувати логарифми з десятковою основою або натуральні логарифми.

Біофізичні шкали призначені для відображення даних про реакцію організму живої істоти на вплив фізичного або хімічного фактору.

Прикладом може служити шкала доз іонізуючого випромінювання або шкала впливу невагомості.

1.6. Метадані

Поряд з поняттям про дані набуло поширення поняття про метадані. Власне метадані – це дані, які містять загальні характеристики набору даних. Метадані можуть бути представлені у вигляді каталогів, довідників, реєстрів і таке інше.

Метадані можуть містити такі узагальнюючі характеристики даних:

-структура даних,

-зміст даних,

-статус набору,

-походження даних,

-фізичне місцезнаходження,

-формати даних,

-умови доступу до даних,

-авторські права.

Метадані є важливою частиною баз даних. Кожна система управління базами даних має свою структуру та формат метаданих, на основі яких виконуються всі процедури доступу до даних.

Метадані програмних пакетів та сховищ розміщуються в тому ж репозиторії, у якому зберігаються дані. Ці метадані найчастіше використовують при роботі зі сховищами за допомогою різних інструментів. Метадані використовують при застосування сховища для реалізації процесів проєктування та при адмініструванні сховища.

Метадані також використовуються у ВЕБ-програмуванні. Цими даними, як правило, користуються пошукові системи. Дані можуть містити інформацію про власника даних, використану систему кодування та багато інших додаткових параметрів.

Отже, на завершення слід відмітити, що в даному розділі розглянуто тільки основні шляхи створення, класифікації та вимірювання даних. В реальному світі існує величезна кількість наборів даних, які не вкладаються в жодну з даних класифікацій. Існують також набори, які мають часткові властивості кількох розглянутих типів даних чи шкал вимірювання.

Дані відображають інформацію про навколишній світ і тому можуть бути такими ж складними.

Контрольні запитання до розділу 1

1. Назвіть основну та похідні одиниці вимірювання обсягів даних в комп’ютерах та співвідношення між ними.
2. Які категорії даних вам відомі, наведіть приклади даних, які відповідають названим вами категоріям.

3. Назвіть основні типи наборів даних, що розглядалися у даному посібнику. Коротко опишіть кожний з наведених типів наборів даних.
4. Дайте визначення шкали для вимірювання даних. Які шкали вимірювання даних вам відомі?
5. Дайте визначення і наведіть приклади застосування абсолютної шкали вимірювання даних.
6. Що таке метадані і які загальні характеристики даних вони можуть містити?

2. ОГЛЯД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ «ANACONDA»

2.1. Установка дистрибутиву «Anaconda»

Anaconda – дистрибутив для мови програмування Python з відкритим кодом для обробки даних великого обсягу, побудови аналітичних прогнозів і наукових обчислень. Anaconda використовують для того, щоб спростити управління і використання пакетів.

Скачати Anaconda3 для Windows можна за посиланням

https://www.anaconda.com/products/individual

Загальний вигляд стартового вікна показаний на рис. 2.1.

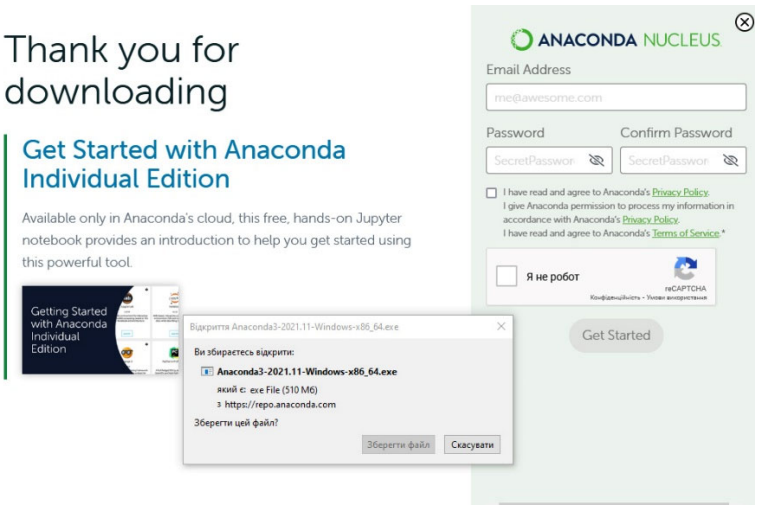


Рис. 2.1. Вікно завантаження «Anaconda»

Anaconda працює також під операційними системами Mac OS X та Linux.

Для цих операційних систем необхідно зайти на сторінку дистрибутивів. На цій сторінці можна вибрати необхідний дистрибутив та скачати його.

Далі розглянемо процес інсталяції дистрибутиву Anaconda3.

2.1.1. Процес установки дистрибутиву «Anaconda»

1. Містить Jupyter notebook та інші популярні інструменти
2. Можливо налаштовувати середовище та створювати канали
3. Можливо встановлювати пакети та керувати контролем версій
4. Містить потужні бібліотеки та інструменти, доступні в Anaconda

Увага!

Для уникнення проблем при використанні Anaconda3 необхідно дотримуватися рекомендацій, які надаються. При інсталяції даного дистрибутиву «за замовчуванням» частина корисних функцій не буде працювати.

1. Скачати дистрибутив Anaconda3-2021.11-Windows-x86_64.exe і запустити його. Вікно (рис. 2.2) одержимо після запуску дистрибутиву:

Рис. 2.2. Стартове вікно інсталяції «Anaconda»

Next натискаємо, впевнившись попередньо, що всі додатки закриті.

2. Наступне вікно, показане на рис. 2.3 – це ліцензійна угода.

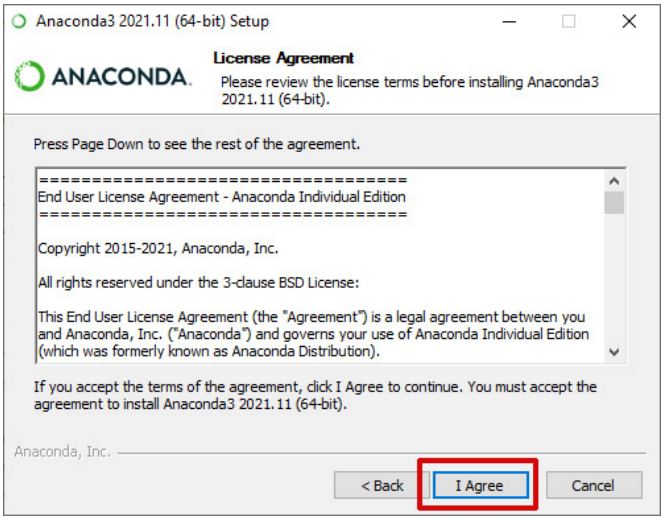


Рис. 2.3. Вікно ліцензійної угоди

Далі натискаємо кнопку «I Agree».

3. Наступний скріншот дозволяє вибрати характер доступу до інтегратора «Anaconda» (рис. 2.4).

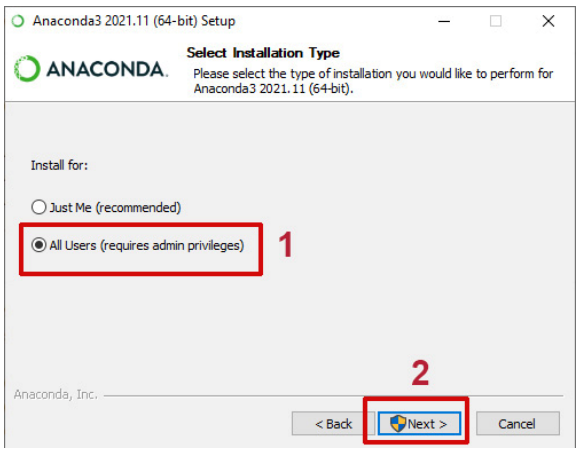


Рис. 2.4. Вибір способу доступу

Під Windows можна вибрати **All Users**, якщо ви маєте права адміністратора. В інших OS вибирайте відповідно до прав доступу. Вибір All Users дозволяє зробити цю програму доступною для всіх користувачів у системі. Тому він є бажаним, якщо ви маєте права адміністратора.

4. За замовчуванням Anaconda буде інстальована в папку ProgramData. Але таке інсталювання може викликати проблеми у подальшій роботі, оскільки папка ProgramData за замовчуванням відкрита тільки для читання. Зміна статусу доступу до папки ProgramData не бажана. Це призведе до зниження рівня безпеки для вашого комп'ютера.

Тому дистрибутив Anaconda можна встановити в попередньо створену папку, наприклад, зі шляхом «C:\Anaconda3» для Windows, як показано на рис. 2.5.

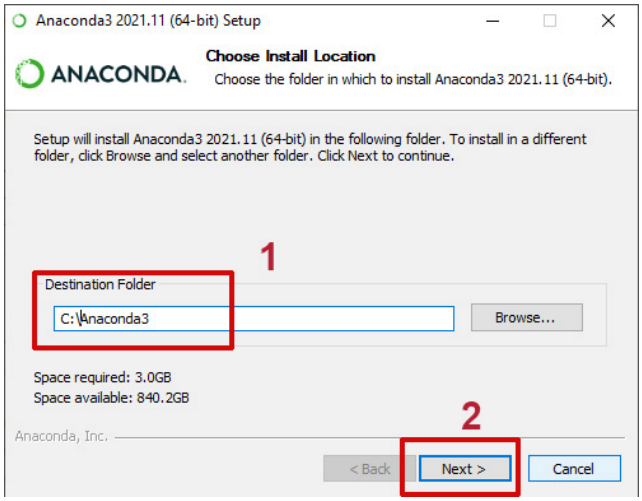


Рис. 2.5. Вибір шляху до розміщення дистрибутиву «Anaconda»

5. В наступному вікні ви маєте відмітити чек-бокси, як зображено на рис. 2.6.

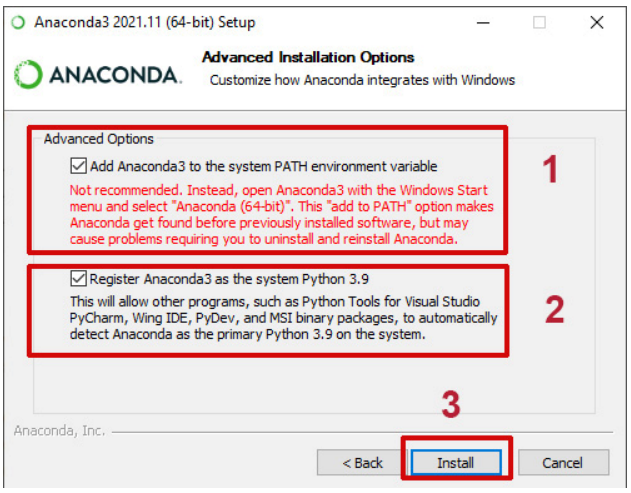


Рис. 2.6. Розширені параметри інсталяції

Це дає можливість завантажувати продукти інтегратора Anaconda з довільної директорії. Така установка накладає на користувача обов'язок попередньо видаляти дану версію програми перед установкою нової версії.

6. Далі починається процес встановлення Anaconda, який може продовжуватися кілька хвилин залежно від потужності комп'ютера. Цей процес відбувається автоматично і не потребує втручання користувача. Процес інсталяції показаний на рис. 2.7.

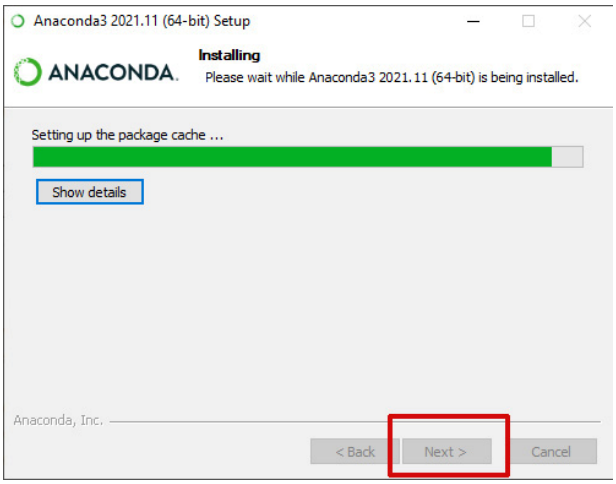


Рис. 2.7. Процес інсталяції

У випадку успішного завершення інсталяції натискаємо «Next».

7. З отриманого вікна (рис. 2.8) можливо безпосередньо перейти до встановлення IDE PyCarm Professional. Порядок установки цього програмного забезпечення розглянемо окремо.

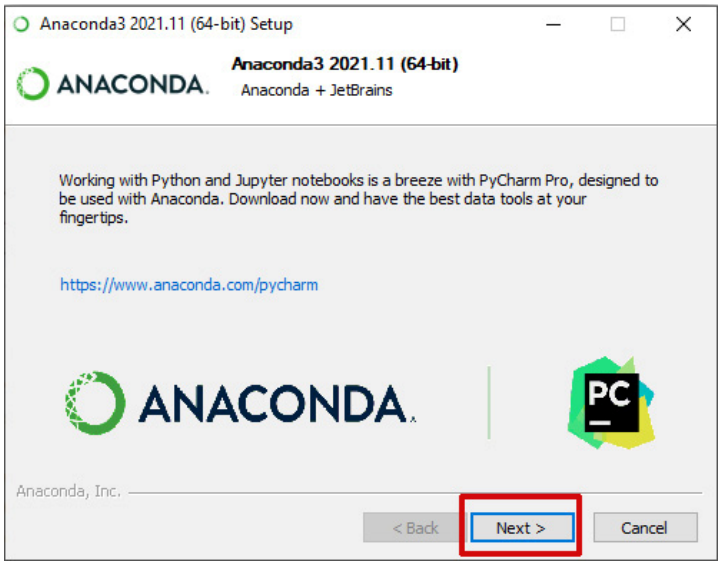


Рис. 2.8. Вікно установки PyCharm

9. Після вказаних на скріншоті дій ви переходите на Tutorial Anaconda (Рис. 2.9).

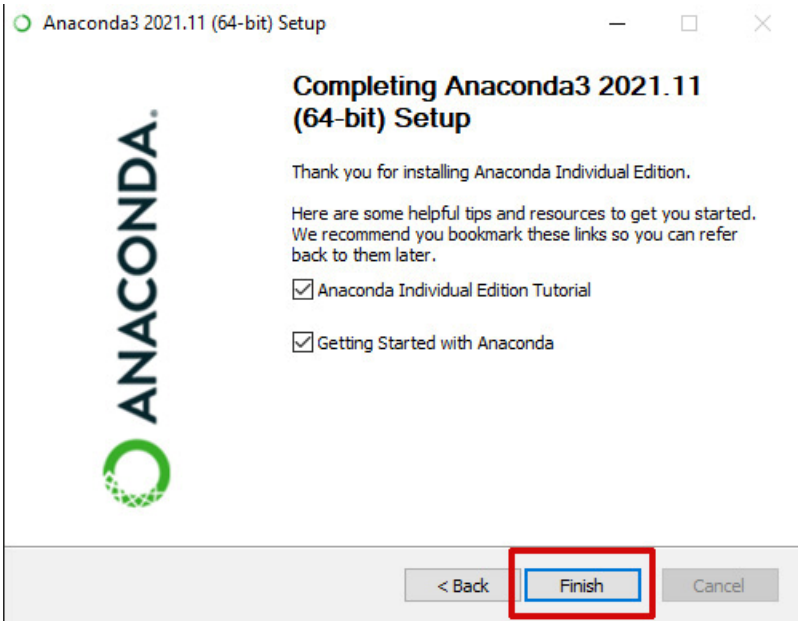


Рис. 2.9. Заключне вікно

2.1.2. Робота з навігатором «Anaconda»

Anaconda Navigator – це десктоп графічного інтерфейсу користувача (GUI), включений у дистрибутив Anaconda3.

Він дозволяє

-запускати програми,

-керувати пакетами conda,

-керувати середовищами,

-керувати каналами.

Всі ці дії можна виконувати використанням команд командного рядка.

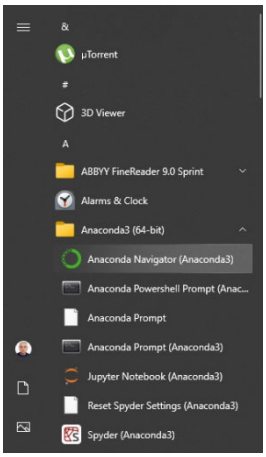
Navigator може шукати пакети:

на Anaconda.org

в локальному репозиторії Anaconda.

Він доступний для Windows, macOS та Linux.

Запуск Anaconda Navigator



У меню «Пуск», яке показане на рис. 2.10, виберіть настільний додаток Anaconda Navigator.

Рис. 2.10. Меню «Пуск» ОС «Windows 10»

Керування середовищами

Створимо нове середовище з ім'ям **fictenv** і встановимо в нього пакет:

1. У Навігаторі перейдіть на вкладку **Environments**, потім натисніть кнопку **Create**. З'явиться діалогове вікно **«Create new environment»**.

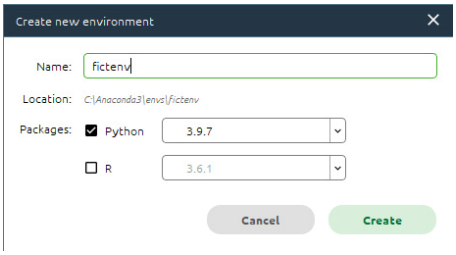


Рис. 2.11. Вікно створення середовища

2. У полі **Name** середовища введіть описову назву вашого середовища. У полі **Python** виберіть версію мови, як показано на рис. 2.11.

3. Натисніть **Create**. Навігатор створює нове середовище та активує його (рис. 2.12).

Тепер маємо два середовища:

-базове середовище base(root)

-наше середовище fictenv.

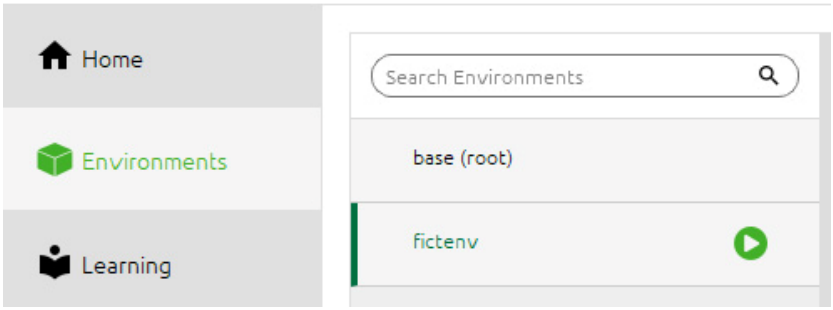


Рис. 2.12. Активація середовища

4. Перемикання між ними (активація та деактивація середовища), здійснюється натисканням на назву середовища, яке ви хочете використовувати.

Керування пакетами

1. Щоб знайти вже встановлений пакет, виберіть назву середовища, у якому потрібно шукати. Встановлені пакети відображаються на правій панелі (рис. 2.13).

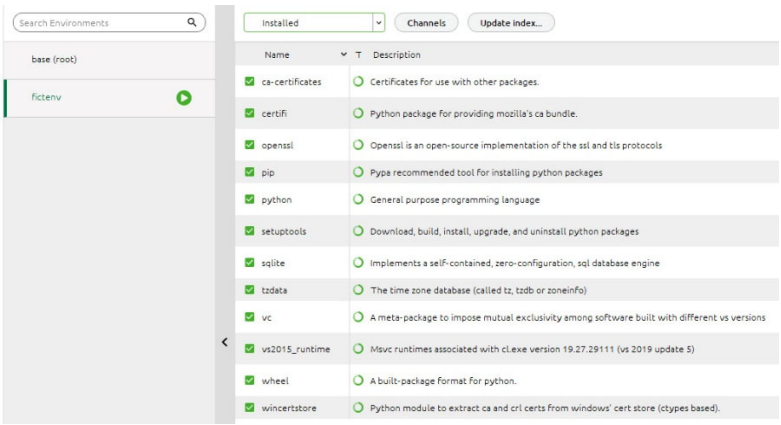


Рис. 2.13. Перегляд встановлених пакетів

2.2. Робота з Jupyter notebook

Вибираємо в меню Windows пункт Anaconda Navigator. Користуючись інтерфейсом Anaconda, запускаємо «Launch» для Jupyter notebook.

У результаті відкриваємо вікно Jupyter такого вигляду, як показано на рис. 2.14.

Рис. 2.14. Вікно керування зошитами Jupyter

Для використання інтерпретатора Python в Jupyter notebook натискаємо на кнопку «New» і вибираємо пункт меню Python3.

Запуск інтерпретатора Python в Jupyter notebook відбувається шляхом натискання кнопки «Run», як показано на рис. 2.15:

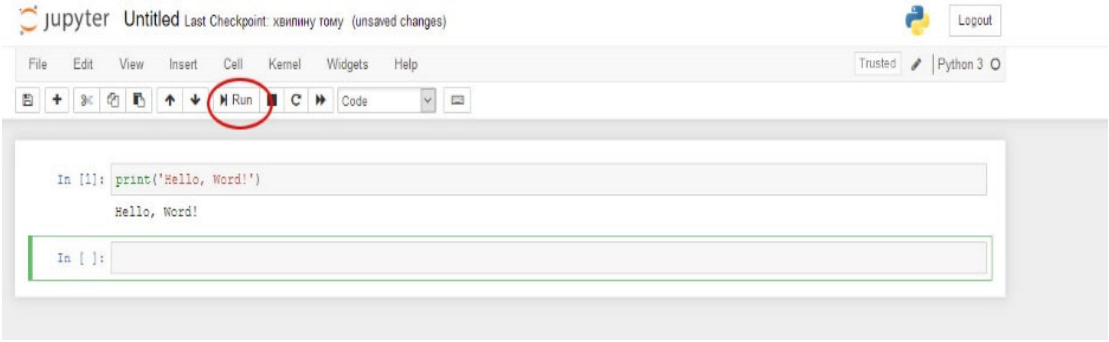


Рис. 2.15. Запуск інтерпретатора

2.2.1. Створення зошита в Jupyter notebook

1. Створимо папку з довільною назвою.

2. Перейдемо у дану папку та викличемо звідти консоль Windows за допомогою команди «cmd», як показано на скріншоті (рис. 2.16).

Рис. 2.16. Запуск термінала

3. У відкритому вікні консолі викликаємо Jupyter notebook.

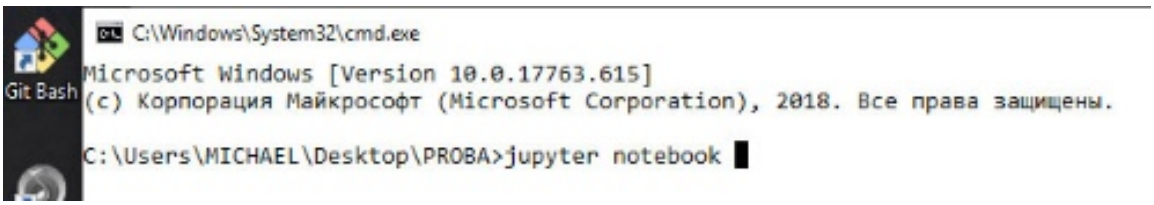


Рис. 2.17. Вікно термінала

4. В результаті відкривається пустий зошит Jupyter.

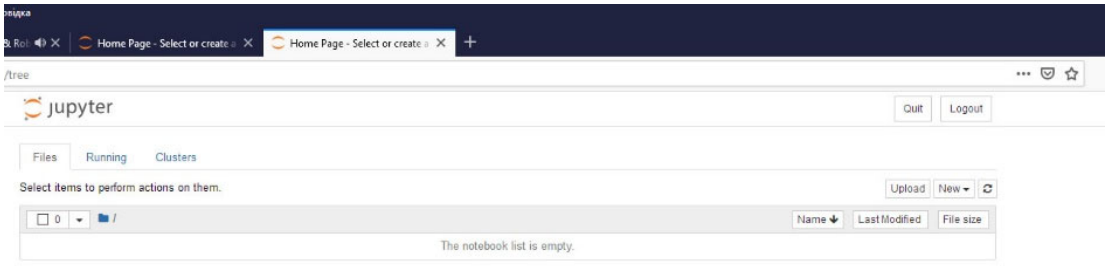


Рис. 2.18. Вікно каталогу Jupyter

5. Результати роботи, наприклад, створені програми, можна зберегти у файлі, вибравши пункт меню File (рис. 2.19).

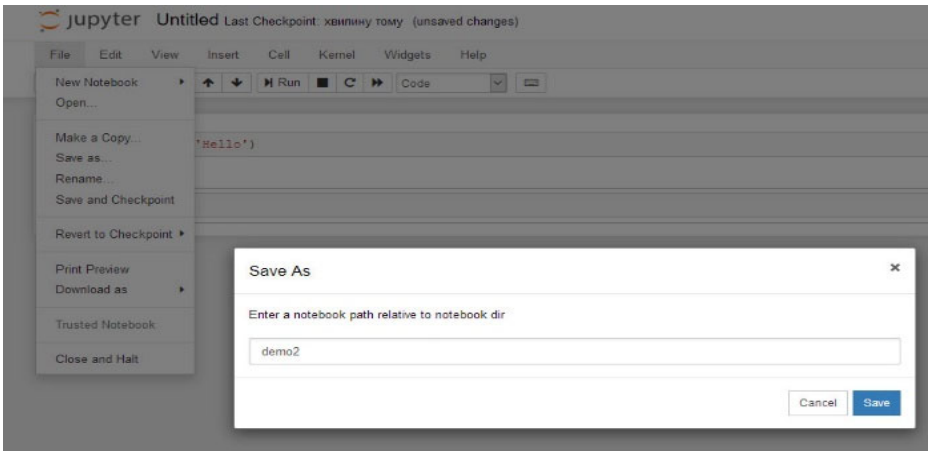


Рис. 2.19. Збереження зошита Jupyter

6. Файл demo2 буде збережено у відповідній папці (рис. 2.20)

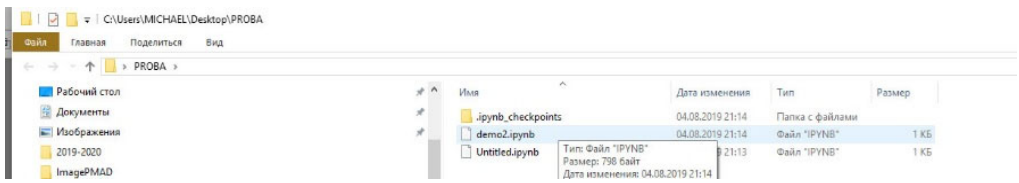


Рис. 2.20. Збережений зошит

Jupyter notebook дає можливість розробляти, документувати та запускати програми мовою Python, він складається з двох компонентів: веб-додаток, що запускається в браузері, і ноутбуки – файли, в яких можна працювати з початковим кодом програми, запускати його, вводити і виводити дані і т. ін.

2.2.2. Приклади роботи з Jupyter notebook

Розглянемо кілька прикладів, які дають можливість зрозуміти принципи роботи з Jupyter notebook.

Запустимо Jupyter notebook, користуючись попередніми інструкціями, і створимо папку для прикладів.

Створення папки

Для створення папки натискаємо на New у правій частині екрана і вибираємо у відповідному меню пункт Folder (рис. 2.21).

Рис. 2.21. Створення папки для зошита

За замовчуванням папці присвоюється ім'я "Untitled folder".

Назву папки необхідно змінити. Для цього поставимо позначку навпроти імені папки і натиснемо на кнопку "Rename" зліва (рис. 2.22).

Рис. 2.22. Перейменування папки для зошитів

Заходимо в цю папку і створюємо в ній зошит. Для цього, як раніше описувалось, скористаємось кнопкою New з наступним вибором "Python 3" у меню (рис. 2.23).

Рис. 2.23. Створення зошита для внесення коду мовою Python

Заповнення зошита

Загальний вигляд зошита, який містить одну комірку (рис. 2.24)

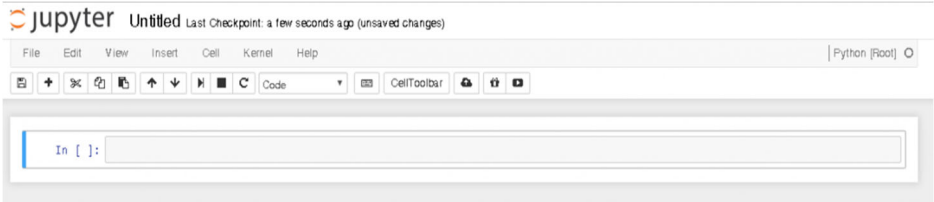


Рис. 2.24. Комірка в зошиті Jupyter

Код мовою програмування Python або текст в нотації Markdown потрібно вводити в комірки (рис. 2.25):

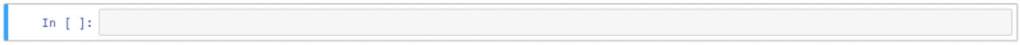


Рис. 2.25. Окремий вигляд комірки

Якщо необхідно вводити код Python, то на панелі інструментів необхідно виставити властивість “Code” (рис. 2.26).

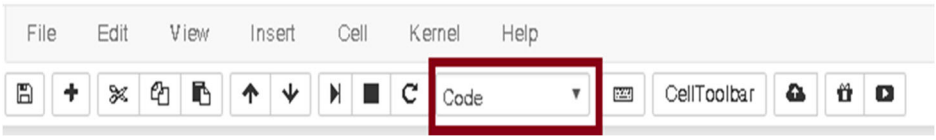


Рис. 2.26. Вибір властивості комірки для вводу коду

Якщо вводимо Markdown текст, то выставляємо “Markdown” (рис. 2.27).

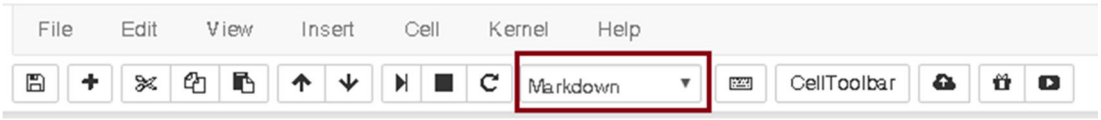


Рис. 2.27. Вибір властивості комірки для вводу тексту

Як приклад, запишемо просту арифметичну задачу:

-виставляємо властивість "Code",

-вводимо в комірку "2 + 3" без лапок і натискаємо

- Ctrl + Enter, введений нами код буде виконаний інтерпретатором Python,

-Shift + Enter буде виконано код і створено нову комірку, яка буде розташована рівнем нижче, так, як показано на скріншоті (рис. 2.28).

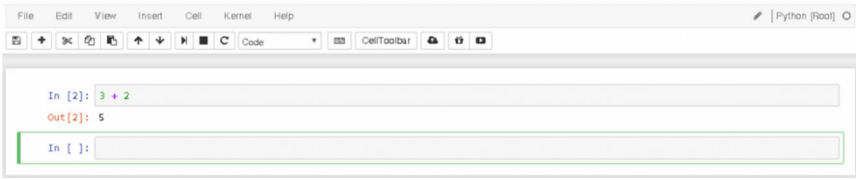


Рис. 28. Приклад виконання коду

Виконайте ще кілька прикладів (рис. 2.29):

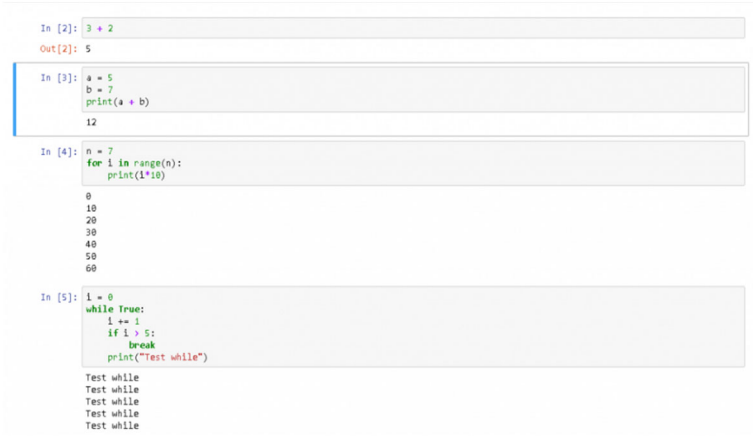


Рис. 2.29. Приклади роботи з зошитами

2.2.3. Основні елементи інтерфейсу Jupyter notebook

У кожного ноутбука є ім'я, воно відображається у верхній частині екрана. Для зміни імені натисніть на його поточне ім'я і введіть нове (рис. 2.30).

Рис. 2.30. Введення назви зошита

Елементи інтерфейсу включають панель меню (рис. 2.31):

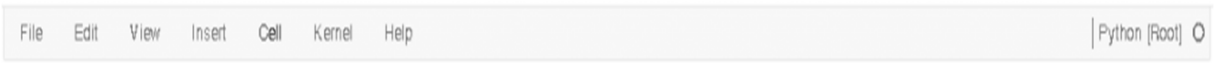


Рис. 2.31. Меню Jupyter Notebook

панель інструментів (рис. 2.32):

Рис. 2.32. Панель інструментів Jupyter Notebook

і робоче поле з комірками (рис. 2.33):

Рис. 2.33. Робоче поле з коміркою

Ноутбук може перебувати в одному з двох режимів – це режим редагування (Edit mode) і командний режим (Command mode).

Поточний режим відображається на панелі меню в правій частині, в режимі редагування з'являється зображення олівця, відсутність цієї іконки означає, що ноутбук знаходиться в командному режимі (рис. 2.34).

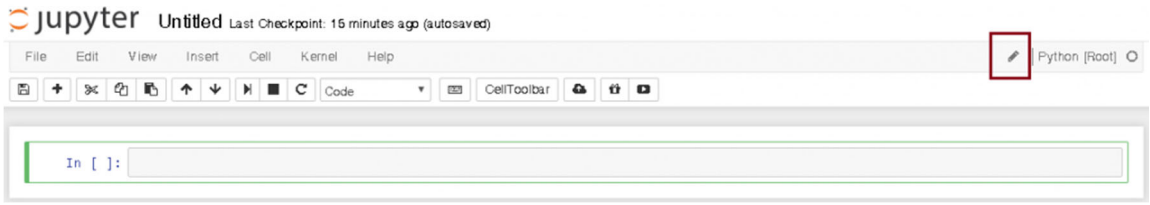


Рис. 2.34. Позначення командного режиму

Щоб відкрити текст довідки по сполученнях клавіш, потрібно натиснути "Help-> Keyboard Shortcuts" (рис. 2.35).

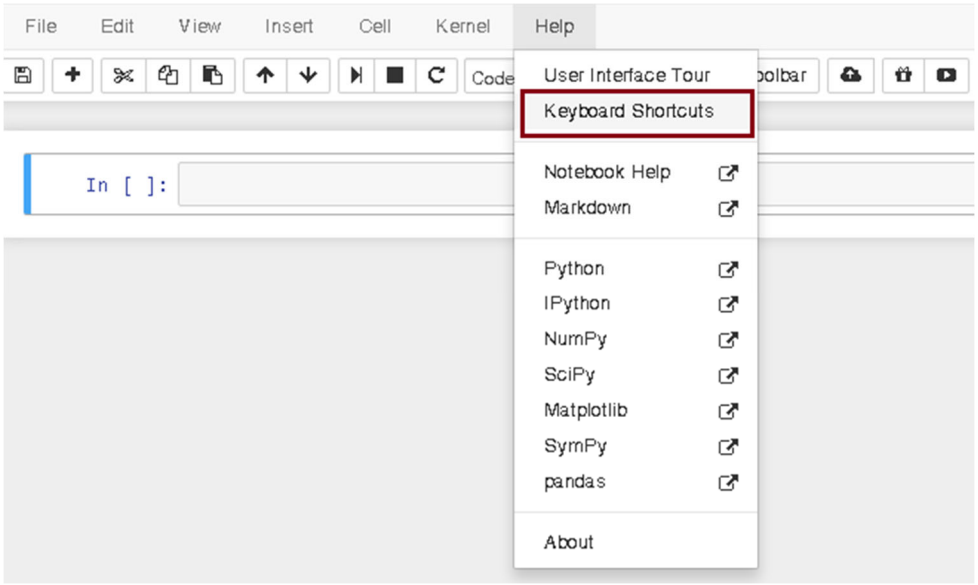


Рис. 2.35. Меню довідки

У крайній правій частині панелі меню знаходиться індикатор завантаженості ядра Python. Якщо ядро знаходиться в режимі очікування, то індикатором є коло (рис. 2.36).

Рис. 2.36. Режим очікування

Якщо ядро виконує завдання, то зображення зміниться на зафарбований круг (рис. 2.37).

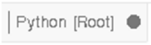


Рис. 2.37. Режим роботи

2.2.4. Запуск і переривання виконання коду

Якщо ваша програма зависла, то можна перервати її виконання, вибравши на панелі меню пункт Kernel -> Interrupt.

Для додавання нової комірки використовуйте Insert-> Insert Cell Above та Insert-> Insert Cell Below.

Для запуску комірки використовуйте команди з меню Cell, або «швидкими» клавішами:

Ctrl + Enter – виконати вміст комірки.

Shift + Enter – виконати вміст комірки і перейти на комірку нижче.

Alt + Enter – виконати вміст комірки і вставити нову комірку нижче.

2.2.5. Доступність зошитів для інших користувачів

Існує кілька способів поділитися своїм ноутбуком з іншими користувачами, причому так, щоб їм було зручно з ним працювати:

• передати безпосередньо файл ноутбука, який має розширення ".ipynb", при цьому відкрити його можна тільки за допомогою Jupyter Notebook;

• конвертувати ноутбук в html;

• використовувати https://gist.github.com/;

• використовувати http://nbviewer.jupyter.org/.

2.2.6. Вивід зображень у зошиті

Друк зображень може стати в нагоді в тому випадку, якщо ви використовуєте бібліотеку matplotlib для побудови графіків. За замовчуванням графіки

не виводяться в робоче поле ноутбука. Для того, щоб графіки відображалися, необхідно ввести і виконати наступну команду:

```
% Matplotlib inline
```

Приклад виведення графіка представлений на скріншоті нижче (рис. 2.38).

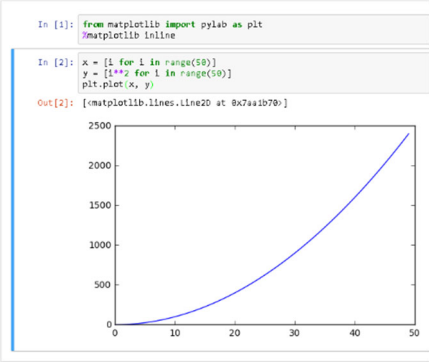


Рис. 2.38. Приклад відображення графіка

Приклад створення власного зошита

1. Запускаємо Jupyter notebook в папці проекту
2. Створюємо новий зошит шляхом вибору в меню New->Python 3
3. Перейменовуємо зошит, виконуючи клік на untitled
4. Клікаєм на нашу першу комірку
5. У випадаючому меню замінюємо Code на Markdown
6. Знову активізуємо комірку кліком (зелена смужка зліва) і пишемо:

```
# (пробіл)Прізвище Ім'я По батькові
```

та натискаємо Shift+Enter

7. Замінюємо Code на Markdown, знову активізуємо комірку кліком і пишемо:

```
# (пробіл) Група ІО-91
```

та натискаємо Shift+Enter

8. У третій комірці пишемо:

print('Hello, World!!')

і натискаємо Ctrl+Enter

9. Отримуємо приблизно такий результат (2.39):

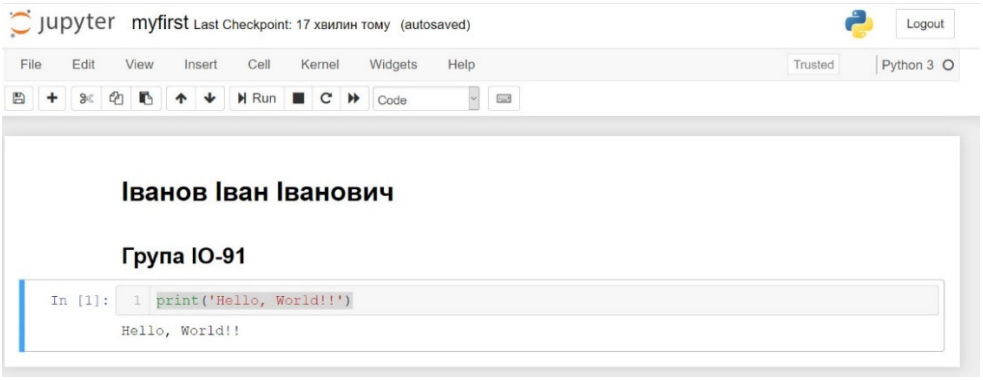


Рис. 2.39. Перший зошит Jupyter

10. Зберігаємо свій перший зошит.

Контрольні запитання до розділу 2

1. Вкажіть призначення дистрибутиву «Anaconda». Під якими операційними системами може працювати дистрибутив «Anaconda»?

2. Які чек-бокси необхідно відмітити у вікні інсталяції «Advanced Installation Options» і яку додаткову функціональність ви після цього отримаєте?

3. Чи існує можливість завантаження додаткового IDE в процесі інсталяції дистрибутиву «Anaconda»?

4. Продемонструйте послідовність дій, які небхідно виконати для завантаження «Anaconda Navigator».

5. Поясніть процес створення нового середовища «Anaconda», активізацію необхідного середовища та керування пакетами в активному середовищі.

6. Продемонструйте дії, які необхідно виконати для запуску «Jupyter Notebook» в довільній папці. Створіть зошит, який містить комірки з відомими вам типами даних.

3. ОГЛЯД АЛГОРИТМІЧНОЇ МОВИ PYTHON

3.1. Огляд роботи зі змінними

3.1.1. Правила іменування змінних

Необхідно дотримуватись таких правил іменування змінних:

1. Кожна змінна повинна мати *унікальне ім'я*, що складається з латинських букв, цифр і знаків підкреслення.

Приклади правильного задавання змінних:

```
A,  a,  bin,  bin124,  rim_12,  ALPHA_2017
```

2. Ім'я змінної *не може починатися* з цифри.

1A , 5myData

3. Слід *уникати символу підкреслення* на початку імені,

_velocity , _pressure123

оскільки ідентифікатори з таким символом мають спеціальне призначення.

4. Як ім’я змінної *не можна використовувати ключові слова*. Одержати список усіх ключових слів дозволяє код, наведений у прикладі 3.1.

Приклад 3.1. Список усіх ключових слів

```
>>> import keyword
>>> keyword.kwlist
 ['False',  'None',  'True',  'and',  'as',  'assert',
'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']
```

5. Слід *уникати збігів із вбудованими ідентифікаторами*.

Приклад 3.2. Одержання списку вбудованих ідентифікаторів

```
>>> import builtins
>>> dir(builtins)
```

```
['Arithmeticerror', 'Assertionerror', 'Attributeerror',
'Baseexception', 'Blockingioerror',…….. str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

Рекомендовано дотримуватись:

1. Не використовувати кирилицю при назві змінних.

~~Скорость~~ , ~~п_125~~

2. Назва змінної має обов'язково нести змістовне навантаження.

```
my_age = 17, my_height = 175
```

В імені змінної важливо враховувати регістр букв: x і X – різні змінні:

Приклад 3.3.

```
>>> x = 10; X = 20
>>> x, X
(10, 20)
```

3.1.2. Типи даних

Логічний тип

bool – логічний тип даних (числовий тип)

Може містити значення True або False, які поводяться як числа 1 і 0 відповідно.

```
>>> type(True), type(False)
(<class 'bool'>, <class 'bool'>)

>>> int(True), int(False)
(1,0)
```

Логічні змінні також можуть використовуватися в логічних виразах (наприклад, в умовних операторах). Python очікує, що логічний вираз після виконання в результаті дає логічне значення. Це можна пояснити на прикладах.

Приклад 3.4.

```
size=int(input("write the number"))
if size >= 0:
```

```
    print('number positive')
else:
    print('number negative')
```

У режимі інтерпретатора ми можемо побудувати логічний вираз і одержимо в результаті false або true
Приклад 3.5.
`>>> size = 1`
`>>> size < 0`
`False`
`>>> size = 0`
`>>> size < 0`
`False`
`>>> size = -1`
`>>> size < 0`
`True`
	Через деякі старі проблеми, що залишилися від Python 2, логічні значення можна розглядати як числа. True це 1; False це 0.
	Приклад 3.6.
	`>>> True + True`
	`2`
	`>>> True - False`
	`1`
	`>>> True * False`
	`0`

Цілочисельний тип `int` *(числовий тип)*

Це тип для цілих чисел. Розмір числа обмежений лише обсягом оперативної пам'яті:

```
>>>type(2147483647),type(99999999999999999)
(<class 'int'>, <class 'int'>)
```

Як правило, тип **int** представляє числа в діапазоні

$-2\ 147\ 483\ 648\ (-2^{31})$ до $2\ 147\ 483\ 647\ (2^{31}-1)$

Для задавання значення змінної типу **int** можна використовувати системи числення з основами

2, 8, 10, 16.

Приклад 3.7. Введення літералів у різних системах числення.

```
# Десяткове число – (0,1,2,3,4,5,6,7,8,9)
#Послідовність чисел:
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
```

```
>>> 123445567788778
123445567788778
```

Двійкове число (починається з префікса 0b)-(0,1)

#Послідовність чисел:

```
 0,1,10,11,100,101,110,111,1000,1001,1010,1011,1100
>>> 0b00110101101010101010101010101
28136789
```

Вісімкове число (починається з префікса 0o)-

```
(0,1,2,3,4,5,6,7)
```

#Послідовність чисел:

```
0,1,2,3,4,5,6,7,10,11,12,13,14,15,16,17,20,21,22,23…
>>> 0o1237645
343973
```

Шістнадцяткове число (починається з префікса `0x`)

```
(0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)
```

#Послідовність чисел: `0,1,2,3,4,5,6,7,8,9,`

```
A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1
E,1F,20,21,22,23,24,25,26,27,28,29,3A,…
>>> 0xDEC1F
912415
```

Тип з плаваючою точкою `float` *(числовий тип)*

Представляє числа з плаваючою точкою подвійної точності

```
>>> type(5.1), type(8.5e-3)
(<class 'float'>, <class 'float'>)
```

Числа записуються з десятковою точкою або в експонентній формі запису.

0.0, 4., 5.7, -2.5, 2e3, 8.9e-4

Приклад 3.8.

```
>>> float(5.4)
5.4
>>> float(8.9e-4)
```

```
0.00089
>>> float(8.9e-12)
8.9e-12
```

Тип комплексні числа `complex` *(числовий тип)*

Комплексні числа задають у форматі:

```
<Дійсна частина> + <Уявна частина>j
```

Для позначення уявної частини можна використовувати як букву J, так і букву j.

```
>>> type(2+2j)
<class 'complex'>
```

Приклад 3.9.

```
>>> 2+5J
(2+5j)
>>> 23j
23j
>>> 11+3j, 23+12j
((11+3j), (23+12j))
```

Тип `NoneType` *(Спеціальний тип)*

Це об'єкт зі значенням None.

Він позначає відсутність будь-якого значення.

```
>>> type(None)
<class 'NoneType'>
```

У логічному контексті значення None інтерпретується як `False`:

```
>>> bool(None)
False
```

Приклад 3.10.

Можна створити змінну типу `NoneType`:

```
my_variable = None
```

Перевірити:

```
if my_variable is None:
    print('my_variable is None')
```

```
else:
    print('my_variable is not None')
```

Тип `Unicode-Рядки str` *(Послідовність)*

Рядки використовують для запису текстової інформації. Тип **`str`** відноситься до групи «Послідовності».

```
>>> type ("Рядок")
<class 'str'>
```

Приклад 3.11.

```
>>> S = "Spam"
>>> len(S)     #довжина
4
>>> S[0]       #перший елемент в S
"S"
>>> S[1]    #другий елемент зліва
"p"
```

Індекс	0	1	2	3
Рядок	S	p	a	m

Типи `bytes i bytearray` *(Послідовність)*

- `bytes` – незмінювана послідовність байтів:

```
>>> type(bytes("Рядок", "utf-8"))
<class 'bytes'>
```

- `bytearray` – змінювана послідовність байтів:

```
>>> type(bytearray("Рядок", "utf-8"))
<class 'bytearray'>
```

Порівняємо близькі типи

1. тип `str` – об'єкт, що є послідовністю символів (тобто складається з рядків символів в юнікоді довжини 1)

2. типи `bytes` і `bytearray` – об'єкти, що складаються з послідовності цілих чисел (від 0 до 255)

Тип `list` *– список (Послідовність)*

Тип даних `list` подібний до масивів в інших мовах програмування:

```
>>> type( [1, 2, 3] )
<class 'list'>
```

Варіанти задавання списку за допомогою літералів і генераторів.

Приклад 3.12.

```
>>> list1=[1,2,3,4]
>>> print(list1)
[1, 2, 3, 4]
>>> list2 = [x**2 for x in range(10) ]
>>> print(list2)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> list3 = list("abcde")
>>> print(list3)
['a', 'b', 'c', 'd', 'e']
```

Тип `tuple` *— кортеж (Послідовність)*

```
>>> type ( (1, 2, 3) )
<class 'tuple'>
```

Константна послідовність (різнорідних) об'єктів. Зазвичай записують в круглих дужках.

Приклад 3.13.

```
>>> for s in ("one","two"): print(s)
one                 #цикл за значеннями кортежу
two
>>> p = (1.2,3.4,0.9)#точка в тривимірному просторі
>>> print(p)
(1.2, 3.4, 0.9)
>>> p1 = 1, 3, 9       #без дужок
>>> print(p1)
(1, 3, 9)
```

```
>>> p2 = 1, 2, 3, 4,     #кома ігнорується
>>> print(p2)
(1, 2, 3, 4)
```

Тип `range` *– діапазони (Послідовності)*

```
>>> type(range(1))
<class 'range'>
```

Ітератор діапазону має три параметри, але обов'язковим є лише один параметр. Це другий параметр.

Приклад 3.14.

```
>>>a=iter(range(3))
>>>next(a)
0
>>>next(a)
1
>>>next(a)
2
>>>next(a)
Error: StopIteration
```

У цьому прикладі маємо два параметри. Перший параметр задає початок послідовності, а другий параметр на одиницю більший, ніж кінець послідовності.

Приклад 3.15.

```
>>>a=iter(range(3,7))
>>>next(a)
3
>>>next(a)
4
>>>next(a)
5
>>>next(a)
6
```

```
>>>next(a)
Error: StopIteration
```

Цей приклад демонструє використання трьох параметрів. Перший параметр задає початок послідовності, другий параметр на одиницю більший за кінець послідовності, а третій параметр визначає крок послідовності.

Приклад 3.16.

```
>>>a=iter(range(3,10,2))
>>>next(a)
3
>>>next(a)
5
>>>next(a)
7
>>>next(a)
9
>>>next(a)
Error: StopIteration
```

Початок послідовності може бути більшим, ніж її кінець. У цьому випадку використовуємо від’ємнй крок.

Приклад 3.17.

```
>>>a=iter(range(3,-6,-2))
>>>next(a)
3
>>>next(a)
1
>>>next(a)
-1
>>>next(a)
-3
```

```
>>>next(a)
-5
>>>next(a)
Error: StopIteration
```

Тип `dict` – *словники*

Тип даних `dict` подібний до асоціативних масивів в інших мовах програмування:

```
>>> type( {"x": 5, "y": 20} )
<class 'dict'>
```

Словник – це структура даних для зберігання пар

ключ:значення, де значення однозначно визначається ключем. Як ключі можуть використовуватись змінні типу числа, рядка, кортежу й т. п.

Порядок пар ключ-значення довільний.

Приклад 3.18.

```
dan={'Name':'Ivan','Age':17,'Course':1}
print(dan)
print('Ім’я: ',dan['Name'])
------
{'Course': 1, 'Age': 17, 'Name': 'Ivan'}
Ім’я:  Ivan
```

Тип `set` *і* `frozenset` – *множини* **(колекції унікальних об'єктів):**

```
>>>type( {"a", "b", "c"})
<class 'set'>
```

– змінювані множини

```
>>>type(frozenset(["a", "b", "c"]))
<class 'frozenset'>
```

– незмінювані множини

Об'єкт `set` є змінюваним неупорядкованим набором незмінних значень.

Загальні області застосування включають:

тестування членства, видалення дублікатів з послідовності і обчислення математичних операцій, таких як перетин, об'єднання, різниця й симетрична різниця.

Об'єкт `frozenset` не можна змінити під час виконання програми.

Приклад 3.19.

```
    Group={"Galenko","Petrenko","Sydorko"}
Total=len(Group)
print(Group)
print("Total: ",Total)
if " Petrenko " in Group:
      print("Petrenko is a student")
Group.add("Trump")
print(Group)
```

```
    ------
    {'Galenko', 'Sydorko', 'Petrenko'}
    Total:  3
    Petrenko is a student
    {'Galenko', 'Sydorko', 'Trump', 'Petrenko'}
```

Тип `function, module,` *type:*

```
>>> def func(): pass
>>> type(func)
<class 'function'>
```

module:

```
>>> import sys
>>> type(sys)
<class 'module'>
```

type – тип:

Усі дані в мові Python є об'єктами, навіть самі типи даних!

```
>>> class C: pass
>>> type(C)
<class 'type'>
>>> type(type("мм"))
<class 'type'>
```

3.1.3. Змінювані й незмінювані типи

Змінювані типи: списки, словники й тип `bytearray`.

Приклад 3.20. Зміна елемента списку (тип `list`)

```
>>> arr = [1,2,3]
>>> arr[0]=0 #змінюємо перший елемент списку
>>> arr
[0, 2, 3]
```

Незмінювані типи: числа, рядки, кортежі, діапазони й тип `bytes`. Для додавання двох рядків використовуємо операцію *конкатенації*, а посилання на новий об'єкт присвоюємо змінній.

Приклад 3.21.

```
>>> str1 = "авто"
>>> str2 = "транспорт"
>>> str3 = str1 + str2      # Конкатенація
>>> print(str3)   #автотранспорт
```

3.1.4. Послідовності й відображення

Послідовності: рядки, списки, кортежі, діапазони, типи `bytes` і `bytearray`.

Відображення: словники.

Послідовності й відображення підтримують механізм ітераторів, що дозволяє зробити обхід усіх елементів за допомогою функції `next()`.

Приклад 3.22. Вивести елементи списку можна так:

```
>>> arr = [1, 2, 3]
>>> i = iter(arr)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
```

Механізм ітераторів для словника (тип `dict`)

На кожній ітерації повертається ключ:

Приклад 3.23.

```
    d = {"x": 1, "y": 2}
i = iter(d)
c = next(i)
print(c,':',d[c])
c = next(i)
print(c,':',d[c])
    -----
    x : 1
    y : 2
```

Цей механізм майже не використовують у такому вигляді. Частіше його використовують опосередковано в операторі циклу.

Приклад 3.24. Вивід елемента списку:

```
>>> for i in [1, 2]: print(i)
1
2
```

Приклад 3.25. Перебрати слово по буквах:

```
for i in "Рядок": print(i)
```

Приклад 3.26. Перебір елементів словника:

```
>>> d = {"x": 1, "y": 2}
>>> for key in d:
    print(d[key])
1
2
```

3.1.5. Присвоювання значення змінним

У мові Python використовується строга динамічна типізація. Значення змінній присвоюється за допомогою оператора = у такий спосіб:

```
>>> x = 7                #Тип int
```

```
>>> x = 0b11          #Тип int
>>> x = 0o17          #Тип int
>>> y = 7.8            #Тип float
>>> s1 = 'Рядок'  # Тип str
>>> b = True         # Тип bool
```

В одному рядку можна присвоїти значення декільком змінним:

```
>>> x = y = 10
>>> x, y
(10, 10)
```

3.1.6. Особливості групового присвоєння

1. Після присвоювання значення в змінній зберігається посилання на об'єкт, а не сам об'єкт. Це обов'язково слід враховувати при *груповому присвоюванні*.

2. Групове присвоювання можна використовувати для чисел, рядків і кортежів, але для змінюваних об'єктів цього робити не можна.

Приклад 3.27. Чому не можна застосовувати групове присвоєння для змінюваних об'єктів?

```
>>> x = y = [1, 2]      #Наче створили два об'єкти
>>> x, y
([1, 2], [1, 2])
```

У цьому прикладі ми створили список із двох елементів і присвоїли значення змінним `x` и `y`.

Тепер спробуємо змінити значення в змінній `y`:

```
>>> y[1] = 100 # Змінюємо другий елемент
>>> x, y
([1, 100], [1, 100])
```

Як видно з прикладу, зміна значення в змінній `y` привела також до зміни значення в змінній `x`.

Таким чином, обидві змінні посилаються на той самий об'єкт, а не на два різні об'єкти.

Щоб одержати два об'єкти, необхідно робити роздільне присвоювання:

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> y[1] = 100    # Змінюємо другий елемент
>>> x, y
([1, 2], [1, 100])
```

3.1.7. Перевірка на подвійне посилання

Перевірити, чи посилаються дві змінні на той самий об'єкт, дозволяє оператор **is**. Якщо змінні посилаються на той самий об'єкт, то оператор **is** повертає значення True:

Приклад 3.28.

```
>>> x = y = [1,2]  #один об'єкт
>>> x is y
True
>>> x = [1,2]    #різні об'єкти
>>> y = [1,2]    #різні об'єкти
>>> x is y
False
```

3.1.8. Об'єднання посилань при кешируванні

З метою підвищення ефективності коду інтерпретатор виконує кешування малих цілих чисел і невеликих рядків.

Це означає, що якщо ста змінним присвоєне число 2, то в цих змінних буде збережено посилання на той самий об'єкт.

Приклад 3.29.

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
>>> x=123
>>> print(y)
2
```

```
>>>x is y
False
```

3.1.9. Перевірка кількості посилань, позиційне присвоювання

Подивитися кількість посилань на об'єкт дозволяє метод `getrefcount ()` з модуля `sys`:

```
>>> import sys     # Підключаємо модуль sys
>>> sys.getrefcount(2)
304
```

Коли число посилань на об'єкт дорівнює нулю, об'єкт автоматично видаляється з оперативної пам'яті. Виключенням є об'єкти, які підлягають кешируванню.

Крім групового присвоювання, мова Python підтримує *позиційне присвоювання*. У цьому випадку змінні вказуються через кому ліворуч від оператора =, а значення – через кому праворуч.

Приклад 3.30. Позиційне присвоювання:

```
>>> x, y, z = 1, 2, 3
>>> x, y, z
(1, 2, 3)
```

За допомогою позиційного присвоювання можна поміняти значення змінних місцями.

Приклад 3.31.

```
>>> x, y = 1, 2
>>>x, y
(1, 2)
>>> x, y = y, x
>>>x, y
(2, 1)
```

По обидві сторони оператора = можуть бути зазначені послідовності.

Згадаємо, що послідовностями є рядки, списки, кортежі, діапазони, типи `bytes` і `bytearray`.

Приклад 3.32.

```
>>> x, y, z = ”123”       # Рядок
>>> x, y, z
('1', '2', '3')
>>> x, y, z = [1, 2, 3]    # Список
>>> x, y, z
[1, 2, 3]
>>> x, y, z = (1, 2, 3)    # Кортеж
>>> x, y, z
(1, 2, 3)
>>> [x, y, z] = (1, 2, 3)#Список ліворуч, кортеж праворуч
>>> x, y, z
(1, 2, 3)
```

3.2. Оператори розгалуження й цикли в мові Python

3.2.1. Оператор розгалуження `if elif else`

Оператор розгалуження **`if elif else`** дозволяє залежно від значення логічного виразу виконати окрему ділянку програми або, навпаки, не виконати її.

Оператор може використовуватись з різним набором складових залежно від складності умови.

Найпростіший формат оператора **`if`** :

`if` <Логічний вираз>:

$\xrightarrow{4пр}$ <Блок, виконується, якщо умова дійсна>

Блок – *це одна або більше інструкцій мовою Python, які записані підряд з однієї і тієї ж позиції.*

`if` <Логічний вираз>:

$\xrightarrow{4пр}$ <Інструкція 1>

$\xrightarrow{4пр}$ <Інструкція 2>

$\xrightarrow{4пр}$ <Інструкція 3>

Формат оператора **if-else**

if <Логічний вираз>:

$\xrightarrow{4пр}$ < Перший блок, виконується, якщо умова дійсна>

else:

$\xrightarrow{4пр}$ <Другий блок, виконується, якщо умова недійсна>

В розгорнутому вигляді цей варіант оператора **if** має вигляд:

if <Логічний вираз>:

$\xrightarrow{4пр}$ <Інструкція 1>

$\xrightarrow{4пр}$ <Інструкція 2>

$\xrightarrow{4пр}$ <Інструкція 3>

else:

$\xrightarrow{4пр}$ <Інструкція 4>

$\xrightarrow{4пр}$ <Інструкція 5>

$\xrightarrow{4пр}$ <Інструкція 6>

Приклад 3.33.

```
    numvar = int(input("Write the number: "))
if numvar < 20:
     print("floor= ",numvar // 2)
     print("remainder= ",numvar % 2)
     print("second_power= ",numvar**2)
else:
   print ("Your number is greater than 20")
```

```
    Write the number: 5
    floor=  2
    remainder=  1
    second_power=  25
```

Правила для блоків і операторів порівняння.

1. Блоки усередині складної інструкції виділяються шляхом зсуву на однакову кількість пропусків (пробілів). (Зазвичай використовують чотири пробіли).

2. Ознакою кінця блоку є зсув наступної інструкції вліво щодо останньої інструкції блоку (на 4 пробіли).

3. У мові Python логічний вираз не обов'язково брати в дужки, але можна, тому що будь-який вираз може бути розташований усередині круглих дужок.

Проте, круглі дужки слід використовувати тільки за необхідності розмістити умову на декількох рядках.

Для прикладу напишемо програму, яка перевіряє, чи є введене користувачем число парним.

Після перевірки виводиться відповідне повідомлення.

3.2.2. Приклади на правило розміщення блоків

Приклад 3.34.

```
    x = int(input("Введіть число: "))
if x % 2 == 0:
    print(x, " - парне число")
else:
    print(x, " - непарне число")
```

Якщо блок складається з однієї інструкції, то цю інструкцію можна розмістити на одному рядку із заголовком.

Приклад 3.35.

```
    x = int ( input ( "Введіть число: ") )
if x % 2 == 0: print(x, " - парне число")
else: print(x, " - непарне число")
```

3.2.3. Дві інструкції в одному рядку

Якщо в одному рядку розміщено кілька інструкцій, то вони мають відділятися одна від одної крапкою з комою.

Приклад 3.36.

```
    x= int(input("Введіть число: "))
if x % 2 ==0: print(x, end=" "); print("- парне число")
else: print(x, end=" "); print("- непарне число")
```

У цьому прикладі end=" " виводиться для того, щоб уникнути переходу на інший рядок.

ПРИМІТКА

Не розміщуйте дві інструкції в рядку, оскільки подібна конструкція порушує стрункість коду й погіршує його супровід надалі. Завжди розміщуйте інструкцію на окремому рядку, навіть якщо блок містить тільки одну інструкцію.

Наступний код має набагато простіший і приємніший вигляд у порівнянні з попереднім:

Приклад 3.37.

```
    x = int (input ( "Введіть число: ") )
if x % 2 == 0:
     print(x, end=" ")
     print ( "- парне число")
else:
     print(x, end=" ")
     print("- непарне число")
```

3.2.4. Формат оператора `if elif else`

if <Логічний вираз1>:

$\xrightarrow{4пр}$ <Блок виконується, якщо умова1 дійсна>

elif <Логічний вираз2>:

$\xrightarrow{4пр}$ <Блок виконується, якщо умова2 дійсна>

else:

$\xrightarrow{4пр}$ <Блок виконується, якщо всі умови недійсні>

Розгорнутий вигляд оператора **if elif else**:

if <Логічний вираз1>:

$\xrightarrow{4пр}$ <Інструкція 1>

$\xrightarrow{4пр}$ <Інструкція 2>

elif <Логічний вираз2>:

$\xrightarrow{4пр}$ <Інструкція 3>

$\xrightarrow{4пр}$ <Інструкція 4>

else:

$\xrightarrow{4пр}$ <Інструкція 5>

$\xrightarrow{4пр}$ <Інструкція 6>

Приклад 3.38.

```
numvar = int(input("Write the number: "))
if numvar < 20 and numvar % 2 == 0:
 print("This is an even number")
 print("floor by 3 = ",numvar // 3)
 print("second_power = ",numvar**2)
elif numvar < 20 and numvar % 2 !=0:
 print("This is an odd number")
 print("floor by 4 = ",numvar // 4)
 print("3hd power = ",numvar**3)
else:
  print ("Your number is greater than 20")
```

3.2.5. Загальний вигляд оператора if elif else

if <Логічний вираз>:

$\xrightarrow{4пр}$ <Блок виконується, якщо умова дійсна>

[**elif** <Логічний вираз>:

$\xrightarrow{4пр}$ <Блок виконується, якщо умова дійсна>]

[**elif** <Логічний вираз>:

$\xrightarrow{4пр}$ <Блок, виконуваний, якщо умова дійсна>]

……

[**elif** <Логічний вираз>:

$\xrightarrow{4пр}$ <Блок, виконуваний, якщо умова дійсна>]

[**else:**

$\xrightarrow{4пр}$ <Блок, виконуваний, якщо всі умови неправильні>

Примітка. У квадратних дужках відображені не обов'язкові складові оператора розгалуження.

Оператор `if ... else` дозволяє перевірити відразу кілька умов.

Приклад 3.39.

```
print("""Якою операційною системою ви користуєтесь?
1 - Windows 10
2 - Windows 8
3 - Windows 7
4 - Windows XP
5 - Інша""")
os = input("Введіть відповідне число: ")
if os == "1":
    print("Ви вибрали: Windows 10")
elif os == "2":
    print("Ви вибрали: Windows 8")
elif os == "3":
    print("Ви вибрали: Windows 7")
elif os == "4":
    print("Ви вибрали: Windows XP")
elif os == "5":
    print("Ви вибрали: інша")
elif not os:
    print("Ви не ввели число")
else:
    print("Ми не визначили вашу операційну систему")
```

У цьому прикладі використовуються `"""..."""` для того, щоб забезпечити багаторядковий вивід даних.

1. За допомогою інструкції **`elif`** у наведеному прикладі ми визначаємо обране значення й виводимо відповідне повідомлення.

2. Логічний вираз **`elif not os:`** не містить операторів порівняння.

•Такий запис еквівалентний наступному: elif os == "":

•Перевірка на рівність значенню `True` логічного виразу виконується за замовчуванням.

•Оскільки порожній рядок інтерпретується як `False`, ми інвертуємо значення, що повертається, за допомогою оператора `not`.

Вкладені оператори **`if`**

Один оператор розгалуження можна вкласти в інший. У цьому випадку відступ вкладеної інструкції повинен бути у два рази більший

(+4 пропуски для кожної нової вкладеності)

Приклад 3.40.

```
print("""Якою операційною системою ви користуєтесь?
1 - Windows 10
2 - Windows 8
3 - Windows 7
4 - Windows XP
5 - Інша""")
os = input("Введіть відповідне число: ")
if os != "":
    if os == "1":
        print("Ви вибрали: Windows 10")
    elif os == "2":
        print("Ви вибрали: Windows 8")
    elif os == "3":
        print("Ви вибрали: Windows 7")
    elif os == "4":
        print ( "Ви вибрали: Windows XP")
    elif os == "5":
        print("Ви вибрали: інша")
    else:
        print("Мы не визначили вашу систему")
```

```
else:
    print("Ви не ввели число")
    print("""Якою операційною системою ви користуєтесь?
1 - Windows 10
2 - Windows 8
3 - Windows 7
4 - Windows XP
5 - Інша""")
os = input("Введіть відповідне число: ")
if not os:
    print("Ви не ввели число")
else:
    if os == "1":
        print("Ви вибрали: Windows 10")
    else:
        if os == "2":
            print("Ви вибрали: Windows 8")
        else:
            if os == "3":
                print("Ви вибрали: Windows 7")
            else:
                if os == "4":
                    print ( "Ви вибрали: Windows XP")
                else:
                    if os == "5":
                        print("Ви вибрали: інша")
```

Оператор **if ... else** має ще один формат:

<Змінна> **=** <значення> **if** <Умова> **else** <значення>

Приклад 3.41.

```
>>> print("Yes" if 10 % 2 == 0 else "No")
```

```
Yes
>>> s = "Yes" if 10 % 2 == 0 else "No"
>>> s
'Yes'
>>> s = "Yes" if 11 % 2 == 0 else "No"
>>> s
'No'
```

3.2.6. Оператор циклу `for in else`

Припустимо, потрібно вивести всі числа від 1 до 100 по одному на рядок. Звичайним способом довелося б писати 100 рядків коду:

```
print(1)
print(2)
…
print (100)
```

За допомогою циклів ту ж дію можна виконати одним рядком коду:

```
for x in range(1, 101): print(x)
```

Іншими словами, цикли дозволяють виконувати ті самі інструкції багаторазово.

3.2.7. Формат оператора циклу `for`

Цикл **`for`** застосовується для перебору елементів послідовності й має такий формат:

`for` <Поточний елемент> **`in`** <Послідовність>:
$\xrightarrow{4пр}$ <Інструкції усередині циклу>
[**`else:`**
$\xrightarrow{4пр}$ <Блок, виконуваний, якщо не використовувався оператор `break`>]
Оператор `break` «викидає» нас з циклу на наступний оператор.

3.2.8. Конструкції оператора `for`

`for` <Поточний елемент> **`in`** <Послідовність>:
$\xrightarrow{4пр}$ <Інструкції усередині циклу>
[**`else:`** <Блок без `break`>]

•<Послідовність> – об'єкт, що підтримує механізм ітерації. Наприклад: рядок, список, кортеж, діапазон, словник і ін.;

•<Поточний елемент> – на кожній ітерації через цей параметр доступний поточний елемент послідовності або ключ словника;

•<Інструкції усередині циклу> – блок, який буде багаторазово виконуватися;

•якщо усередині циклу не використовувався оператор **break**, то після завершення виконання циклу буде виконаний блок в інструкції **else**.

•<Блок без `break`> – не є обов'язковим.

Перебір по рядку

Приклад 3.42. Програма перебору букв у слові.

```
for s in "Я вчуся програмувати":
 print(s, end=" ")
else:
 print ( "\n Цикл виконаний")
```

Результат виконання:

Я в ч у с я п р о г р а м у в а т и

Цикл виконаний

Перебір у списках і кортежах

Приклад 3.43. Програма перебору елементів списку

```
for x in [ "London","Paris","Washington"]:
 print(x)
```

```
London
Paris
Washington
```

Програма перебору елементів кортежу:

```
for y in ( "Europe", "Asia","Africa" ):
 print (y)
```

```
Europe
Asia
```

```
Africa
```

Перебір у словниках

1. Цикл **for** дозволяє також перебрати елементи словників, хоча словники й не є послідовностями.

Перший спосіб перебору використовує метод **keys()**, що повертає об'єкт **dict_keys,** який містить усі ключі словника.

Приклад 3.44. Використання методу `keys()`

```
>>> arr = {"x": 1, "y": 2, "z": 3}
>>> arr.keys()
dict_keys(['y', 'x', 'z'])
>>> for key in arr.keys():
        print(key, arr[key])
```

Другий спосіб перебору словника.

Просто вказуємо словник як параметр –

на кожній ітерації циклу буде повертатися ключ, за допомогою якого усередині циклу можна одержати значення відповідно до цього ключа.

Приклад 3.45. Перебір словника другим способом

```
arr = {"x": 1, "y": 2, "z": 3}
for key in arr:
     print(key, arr[key])
```

Перебір словника з сортуванням ключів

Елементи словника виводяться в довільному порядку, а не в порядку, у якім вони були зазначені при створенні об'єкта.

Щоб вивести елементи в алфавітному порядку, слід відсортувати ключі за допомогою функції **sorted():**

Приклад 3.46.

```
arr = {"x":1, "y":2, "z":3}
for key in sorted(arr):
    print ( key, arr [ key] )
```

Перебір елементів у складних структурах одного розміру

За допомогою циклу **for** можна перебирати складні структури даних. Як приклад виведемо елементи списку, що містить кортеж та список.

Приклад 3.47.

```
arr = [(1,2),["При","віт"]]  # Список і кортеж
for x in arr:
    print(x)
```

Результат:

```
(1, 2)
['При', 'віт']
```

```
for x, y in arr:
    print(x, y, sep="")
```

Результат:

```
12
Привіт
```

Перебір елементів у складних структурах, що містять об'єкти різного розміру

Приклад 3.48.

```
arr = [[1,2],[3,4,5]]
#Список списків різної довжини
for x in arr: # перебір в одну змінну
        print(x)
```

Результат:

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

```
for x,y in arr: # позиційний перебір
        print(x,y)
```

Результат:

```
1 2
Traceback (most recent call last):
ValueError: too many values to unpack (expected 2)
```

Перебір елементів із зірочкою

Приклад 3.49.

```
arr = [[1,2],[3,4,5]]
#Список списків різної довжини
for x,y,*z in arr: # перебір із зірочкою
      print(x,y,z)
```

Результат:

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

```
arr = [[1,2],[3,4,5]]
#Список списків різної довжини
for x,*y in arr: # перебір із зірочкою
      print(x,y)
```

Результат:

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

Перебір елементів послідовності з модифікацією

Дотепер ми тільки виводили елементи послідовностей. Тепер спробуємо помножити кожний елемент списку на 2:

Приклад 3.50.

```
    s = [1,2,3,4,5,6]
for i in s:
    i=2*i
    print(i)
print(s)
```

```
2
4
6
8
10
```

```
12
[1, 2, 3, 4, 5, 6]
```

Перебір послідовності з використанням `range()`

Спосіб одержання доступу до елементів за допомогою функції **`range()`** шляхом генерації індексів.

Формат функція **`range()`:**

`range` `([<Початок>,] <Кінець> [, <Крок>])`

1. Перший необов'язковий параметр `<Початок>`.

Задає початкове значення.

Якщо параметр не зазначений, то за замовчуванням використовується значення 0.

2. Другий обов'язковий параметр `<Кінець>` указує кінцеве значення на одиницю більше.

3. Третій необов'язковий параметр `<Крок>`

Якщо не зазначений, то використовується значення 1. Функція повертає діапазон – особливий об'єкт, який підтримує ітераційний протокол.

Перебір зі зміною послідовності

Функція **`range()`** всередині циклу **`for`** дозволяє одержати значення поточного елемента. Змінимо послідовність шляхом множення кожного елемента списку на 2.

Приклад 3.51.

```
arr = [1, 2, 3]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)
```

Результат: `[2, 4, 6]`

1. Одержуємо кількість елементів списку за допомогою функції **`len()`** і передаємо результат у функцію `range()`.

2. Функція `range()` повертає діапазон значень від 0 до (**`len(arr)-1`**). На кожній ітерації циклу через змінну **`i`** доступний поточний елемент із діапазону

індексів. Щоб одержати доступ до елемента списку, указуємо індекс усередині квадратних дужок. Множимо кожний елемент списку на 2, а потім виводимо результат за допомогою функції **print().**

Приклад використання функції ***range()*** .

Приклад 3.52. Виведемо числа від 1 до 100:

```
for i in range(1,101 ): print(i)
```

Приклад 3.53. Виведемо числа у зворотному порядку від 100 до 1:

```
for i in range(100, 0, -1): print(i)
```

Можна також змінювати значення не тільки на одиницю.

Приклад 3.54. Виведемо всі парні числа від 1 до 100:

```
for i in range(2, 101, 2): print(i)
```

3.2.9. Функція enumerate

Функція має такий формат:

```
enumerate (<Об'єкт> [, start=0])
```

На кожній ітерації циклу **for** вона повертає кортеж з індексу й значення поточного елемента.

Параметр `start` може задати початкове значення індексу.

Приклад 3.55.

```
#Помножимо на 2 кожний елемент списку, який містить парне число.
arr = [1, 2, 3, 4, 5, 6]
for i, elem in enumerate(arr):
    if elem % 2 == 0:
        arr[i] *= 2
print(arr)
```

Результат: `[1, 4, 3, 8, 5, 12]`

Перебір послідовності за допомогою функції `enumerate()`

Функція `enumerate()` не створює список, а повертає ітератор. За допомогою функції `next ()` можна обійти всю послідовність. Коли перебір буде закінчений, виконується виключення `StopIteration`:

Приклад 3.56.

```
    arr = [9, 21,"Line"]
obj = enumerate(arr, start=123)
print(next(obj))
print(next(obj))
print(next(obj))
```

Результат:

```
(123, 9)
(124, 21)
(125, 'Line')
```

3.2.10. Оператор циклу `while`

Виконання інструкцій у циклі **`while`** триває доти, поки логічний вираз дійсний. Цикл **`while`** має наступний формат:

<Початкове значення>

`while` <Умова>:

<Інструкції>

<Збільшення лічильника>

[**`else:`**

<Блок виконується за умови, що не використовується оператор **`break`**>]

Послідовність роботи циклу `while`*:*

1. Змінній-лічильнику присвоюється початкове значення.

2. Перевіряється умова й, якщо вона істинна, виконуються інструкції всередині циклу, інакше виконання циклу завершується.

3. Змінна-лічильник змінюється на величину, зазначену в параметрі <Збільшення>.

4. Перехід до пункту 2.

5. Якщо всередині циклу не використовувався оператор `break`, то після завершення виконання циклу буде виконаний блок в інструкції `else`. Цей блок не є обов'язковим.

Приклад 3.57.

```
#Виведемо всі числа від 1 до 100, використовуючи цикл while
    i = 1           # <Початкове значення>
while i < 101: # <Умова>
     print(i)    # <Інструкції>
     i += 1      # <Збільшення (приріст)>
```

ПРИМІТКА

Якщо <Збільшення> не зазначене, цикл буде нескінченним. Щоб перервати нескінченний цикл, слід натиснути комбінацію клавіш <Ctrl>+<C>. У результаті генерується виключення `Keyboardinterrupt`, і виконання програми зупиняється. Слід ураховувати, що перервати в такий спосіб можна тільки цикл, який виводить дані.

Приклад 3.58.

```
#Виведемо всі числа від 100 до 1
    i = 100
while i:
     print(i)
     i -= 1
```

1. У цьому прикладі умова не містить операторів порівняння.

2. На кожній ітерації циклу ми віднімаємо одиницю зі значення змінної-лічильника.

3. Як тільки значення буде дорівнювати 0, цикл зупиниться. Згадаємо, що число 0 у логічному контексті еквівалентно значенню **False**, а перевірка на рівність виразу значенню **True** виконується за замовчуванням.

Перебір елементів за допомогою `while`

За допомогою циклу **`while`** можна перебирати й елементи різних структур. Але в цьому випадку слід пам'ятати, що цикл `while` працює повільніше, ніж цикл `for`. Як приклад помножимо кожний елемент списку на 2.

Приклад 3.59.

```
arr = [1, 2, 3]
i, count = 0, len(arr)
while i < count:
    arr[i] *= 2
    i += 1
print(arr)
```

Результат: `[2, 4, 6]`

3.2.11. Оператори `continue` та `break`

Оператор `continue` дозволяє перейти до наступної ітерації циклу до завершення виконання всіх інструкцій усередині циклу. Як приклад виведемо всі числа від 1 до 100, крім чисел від 5 до 10 включно.

Приклад 3.60. Застосування оператора `continue`

```
for i in range(1, 101):
if 4 < i < 11:
  continue #Переходимо на наступну ітерацію циклу
   print(i)
```

Оператор **`break`** дозволяє перервати виконання циклу достроково. Для прикладу виведемо всі числа від 1 до 100 ще одним способом.

Приклад 3.61. Застосування оператора **`break`**

```
i = 1
while True:
if i > 100: break # Перериваємо цикл
   print(i)
i += 1
```

В умові вказано значення **`True`**.

У цьому випадку вираз усередині циклу має виконуватися нескінченно.

Однак використання оператора **`break`** перериває виконання циклу, як тільки буде надруковано 100 рядків.

ПРИМІТКА.

Оператор **break** перериває виконання циклу, а не програми, тобто далі буде виконана інструкція, що слідує відразу за циклом.

Приклад 3.62. Додавання невизначеної кількості чисел

```
print("Введіть слово 'stop' для одержання результату")
s = 0
A="0"
while True:
    x = input("Введіть число: ")
    if x == "stop":
        break   # Вихід із циклу
    else:
        A+="+"+x
        print(A)
    y = int(x)  # Перетворимо рядок у число
    s += y
print(A,"=",s )
```

3.3. Списки, кортежі, множини і діапазони в мові Python

Списки, кортежі, множини та діапазони – це послідовності об'єктів.

Кожний елемент таких послідовностей містить лише посилання на об'єкт – тому вони можуть містити об'єкти довільного типу даних і мати необмежений ступінь вкладеності.

Наприклад:

$$A=\Big[\big[(1,2),3\big],\ \big[\{15,18\},\ \big[\{"d":19\},m\big]\big]\Big]$$

Розглянемо список $L=\big[[1,2,3],4,5\big]$

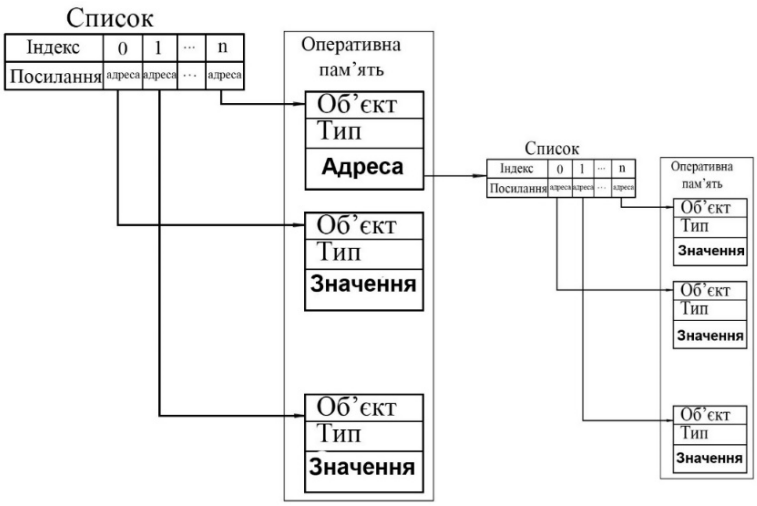


Рис. 3.1. Структура багаторівневого списку

3.3.1. Властивості послідовностей

1. Позиція елемента в списках та кортежах визначається індексом.

СПИСОК	Індекс	0	1	2	3		n
	Значення	Об'єкт	Об'єкт	Об'єкт	Об'єкт		Об'єкт

2. Нумерація елементів у списках, кортежах та діапазонах починається з 0.

3. Списки й кортежі є просто впорядкованими послідовностями елементів.

Як і всі послідовності, вони підтримують такі операції:

- доступ до елемента по індексу,
- одержання зрізу,
- конкатенацію (оператор +),
- повторення (оператор *),
- перевірку на входження (оператор `in`)
- перевірку на невходження (оператор `not in`).

3.3.2. Загальна характеристика списків

1. Списки є змінюваними типами даних.

Це означає, що ми можемо не тільки одержати елемент по індексу, але й змінити його:

Приклад 3.63.

```
>>> arr = [1, 2, 3] # Створюємо список
```

```
>>> arr[0]      # Одержуємо елемент по індексу
1
>>> arr[0] = 50 # Змінюємо елемент по індексу
>>> arr
[50, 2, 3]
```

3.3.3. Створення списку

Створити список можна такими способами:

1. За допомогою функції **list**([<Послідовність>]).

Функція дозволяє перетворити будь-яку послідовність у список. Якщо параметр не зазначений, то створюється порожній список.

Приклад 3.64.

```
>>> list() # Створюємо порожній список
[]
>>> list("String") # Перетворимо рядок у список
['S', 't', 'r', 'i', 'n', 'g']
>>> list ((1, 2, 3, 4, 5))# Перетворимо кортеж у список
[1, 2, 3, 4, 5]
>>> s = {"a", "string", "рядок", 12, 45.123, True}
>>> b = list(s) # Перетворимо множину у список
>>> print(b)
[True, 'string', 12, 45.123, 'рядок', 'a']
>>> list(range(10)) # Перетворимо діапазон у список
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. Створюємо список перерахуванням у квадратних дужках:

Приклад 3.65.

```
>>> arr = [1, "str", 3, "4"]
>>> arr
[1, 'str', 3, '4']
>>> lmy = ["Петренко", "Петро", 22, "роки"]
>>> lmy
```

```
['Петренко', 'Петро', 22, 'роки']
>>> nmy = [12.2, 11.234567, 23e-12]
>>> nmy
[12.2, 11.234567, 2.3e-11]
```

3. Створюємо список застосуванням методу `append()` для заповнення списку поелементно:

Приклад 3.66.

```
>>> arr = [] # Створюємо порожній список
>>> arr.append(1)#додаємо елемент 1(індекс 0)
>>> arr.append("str") # Додаємо елемент "str" (індекс 1)
>>> arr
[1, 'str']
```

Створення списку має такі особливості:

1. При створенні списку в змінній зберігається посилання на об'єкт, а не сам об'єкт.

```
>>> a,b="string",1
>>> my_list=[a,b]
>>> a is my_list[0]
True
>>> b is my_list[1]
True
```

2. Групове присвоювання для списку може бути небажаним.

Приклад 3.67.

```
>>> x = y = [1, 2] # Нібито створили два об'єкти
>>> ligrup=[x, y]
>>> ligrup
[[1, 2], [1, 2]]
>>> ligrup[0] is ligrup[1]
True
```

Список з двох елементів зі значеннями змінних `x` та `y`.

Приклад 3.68.

Тепер спробуємо змінити значення в змінній y:

```
>>> y[1] = 100  # Змінюємо другий елемент
>>> y[1]=100
>>> ligrup
[[1, 100], [1, 100]]
>>> x, y # Змінилося значення відразу у двох змінних
([1, 100], [1, 100])
>>> ligrup[0][0]=600
>>> x[0]
600
>>> y[0]
600
>>> ligrup
[[600, 100], [600, 100]]
```

Щоб одержати два об'єкти, необхідно робити роздільне присвоювання:

Приклад 3.69.

```
>>> x, y = [1, 2], [1, 2]
>>> lidif=[x,y]
>>> lidif
>>> y[1] = 100 # Змінюємо другий елемент
>>> lidif[0] is lidif[1]
False
y[1]=100
lidif
[[1, 2], [1, 100]]
```

КЕШИРУВАННЯ

```
>>> x,y,z=2,2,2 # Слід відрізняти від кеширування
>>> x is y
True
```

```
>>> y=3
>>> x is y
False
```

Особливості створення списку з використанням оператора *

Оператор повторення * діє як групове присвоювання.

Наприклад, у наступній інструкції проводиться спроба створення двох вкладених списків за допомогою оператора *:

Приклад 3.70.

```
>>>arr=[ [1] ] * 2 # Нібито створили два вкладені списки
>>>arr
[[1], [1]]
>>> arr[0] is arr[1]
True
arr[0] [0] is arr[1][0]
True
>>> arr[0].append(5) # Додаємо елемент
>>> arr # Змінилися два елементи
[[1,5], [1,5]].
```

3.3.4. Створення вкладених списків

Створювати вкладені списки слід за допомогою методу `append()` всередині циклу:

Приклад 3.71.

```
>>> arr = []
>>>for i in range(3): arr.append(i)
>>>arr
[0, 1, 2]
>>> arr = []
>>> for i in range(3): arr.append([i])
>>> arr
[[0], [1], [2]]
```

```
>>> arr[0].append(5)
>>> arr
[[0,5], [1], [2]]
```

3.3.5. Створення списків за допомогою генераторів

Генератор списків – спосіб побудувати новий список, застосовуючи вираз до кожного елемента послідовності.

Генератори списків дуже схожі на цикл **`for`**.

Приклад 3.72.

```
>>> arr = [ [i**2] for i in range(3) ]
>>> arr
[[0], [1], [4]]
>>> arr[1].append(2)
>>> arr
[[0], [1, 2], [4]]
>>> arr.append(8)
>>> arr
[[0], [1, 2], [4], 8]
>>> arr[0][0]= 'str'
>>> arr
[['str'], [1, 2], [4], 8]
```

3.3.6. Створення копії списку

Існують три способи створити копію списку:

1. За допомогою функції `list()`.
2. З застосуванням операції добування зрізу.
3. З застосуванням методу `copy()`.

Приклад 3.73. Створення копії за допомогою `list()`

```
>>> x = [1, 2, 3, 4, 5] # Створили списки
>>> y = ["a", "b", "c", "d"]
>>> # створюємо копію списку за допомогою list
```

```
>>> z = list(x)
>>> f = list(y)
>>> z,f
([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd'])
>>> z is x
False
>>> f is y
False
```

Створення копії за допомогою добування зрізу

```
>>> x = [1, 2, 3, 4, 5] # Створили списки
>>> y = ["a", "b", "c", "d"]
>>> z= x[:]
>>> f= y[:]
>>> z,f
([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd'])
>>> f is y
False
>>> z is x
False
```

Створення копії списку викликом методу `copy()`*:*

```
>>> x = ["2016","2017","2018","2019","2020"]
>>> y= x.copy()
>>> y
['2016', '2017', '2018', '2019', '2020']
>>> x is y # Оператор показує, що це різні об'єкти
False
>>> y[4] = "2021" # Змінюємо другий елемент
>>> x,y # Змінився тільки список у змінній y
```

(['2016', '2017', '2018', '2019', '2020'], ['2016', '2017', '2018', '2019', '2021'])

3.3.7. Поверхнева та глибока копії списку

1. Операція присвоювання не копіює об'єкт, вона лише створює посилання на об'єкт.

2. Розглянуті методи створення копії списку створюють поверхневу копію.

Означення поверхневої та глибокої копії

Поверхнева копія створює новий складений об'єкт,

і потім вставляє в нього посилання на об'єкти, що містяться в оригіналі.

Глибока копія створює новий складений об'єкт, і потім рекурсивно вставляє в нього копії об'єктів, що містяться в оригіналі.

Поверхнева копія

Приклад 3.74.

```
>>> x = [1, [2, 3, 4, 5]]# Створили вкладений список
>>> y = list(x)          # Зробили копію списку
>>> x is y               # Різні об'єкти
False
>>>y[0]=200
>>>y[1][1] =100 # Змінюємо елемент
>>> y
[200, [2, 100, 4, 5]]
>>> x# Зміна відбулася і в x!!!
[1, [2, 100, 4, 5]]
```

У прикладі список `y` є поверхневою копією списку `x`:

```
>>> x is y
False
```

Але $x[1][1] == y[1][1] \Rightarrow True$

Отже, `x` і `y` містять однакові посилання на вкладені об'єкти.

Глибока копія

Щоб одержати глибоку копію списку, слід скористатися функцією `deepcopy()` з модуля `copy()` .

Приклад 3.75.

```
>>> import copy # Підключаємо модуль copy
>>> x = [1, [2, 3, 4, 5]]
>>> y = copy.deepcopy(x)#створюємо глибоку копію
>>> y[1] [1] = 100   # Змінюємо другий елемент
>>> x,y  # Змінився тільки список в y
([1, [2, 3, 4, 5]], [1, [2, 100, 4, 5]])
```

Розглянемо приклад, в якому два елементи посилаються на один об'єкт.

Функція `deepcopy()` рекурсивно створює копію кожного вкладеного об'єкта, при цьому зберігаючи внутрішню структуру списку.

Приклад 3.76.

```
>>>import copy   # Підключаємо модуль copy
>>>x = [1, 2]
>>>y = [x, x] # два елементи посилаються на один об'єкт
>>>y
[[1, 2], [1, 2]]
>>>z = copy.deepcopy(y) # Зробили копію списку
>>>z
[[1, 2], [1, 2]]
>>> z[0] is x, z[1] is x, z[0] is z[1]
(False, False, True)
>>> z [0] [0] = 300  # Змінили один елемент
>>> z # Значення змінилося відразу у двох елементах!
[[300, 2], [300, 2]]
>>> x   # Початковий список не змінився
[1, 2].
```

3.3.8. Операції над списками

1. Доступ до елементів списку здійснюється за допомогою квадратних дужок.
2. У квадратних дужках вказується індекс елемента.
3. Нумерація елементів списку починається з нуля.

Приклад 3.77. Вивід елементів списку:

```
>>>r=range(5,20,2)
>>>arr = [r, "str", 4.1, "5"]
>>>arr[0][3]
11
>>>arr.append(10)
>>>arr+=[5,[1, "str",["none"]]
>>>arr
[range(5,20,2),'str',4.1,'5',10,5,[1,'str',['none']]]
```

Позиційне присвоювання для списків

Особливість позиційного присвоювання:

Кількість елементів праворуч і ліворуч від оператора = повинна збігатися, інакше буде виведене повідомлення про помилку:

Приклад 3.78.

```
>>> x, y, z = [1, 2, 3] # Позиційне присвоювання
>>> x, y, z
(1, 2, 3)
>>>x,y,z=[[1,2],[3,4,5],[7]]
>>>x,y,z
([1, 2], [3, 4, 5], [7])
>>> x, y= [1, 2, 3] #Кількість елементів повинна збігатися
Traceback (most recent call last):
  File "<input>", line 1, in <module>
Valueerror: too many values to unpack (expected 2)
```

Зірочка в позиційному присвоюванні

1. В Python 3 при позиційному присвоюванні перед однією зі змінних ліворуч від оператора = можна вказати зірочку (*).

2. У цій змінній буде зберігатися список, що містить «зайві» елементи. Якщо таких елементів немає, то список буде порожнім:

Приклад 3.79.

```
>>>x, y, *z = [[1,2], [3,4,5], [6,7]]
>>>x,y,z
([1, 2], [3, 4, 5], [[6, 7]])
>>>x, y, *z = [[1,2], [3,4,5], [6,7],[8,9]]
>>>x,y,z
([1, 2], [3, 4, 5], [[6, 7], [8, 9]])
>>> x, y, *z = [[1,2], [3,4,5]]
>>>x,y,z
([1, 2], [3, 4, 5], [])
>>> *x, y, z = [[1,2], [3,4,5]]
>>>x,y,z
([], [1, 2], [3, 4, 5])
```

Зірочка повинна бути тільки одна.

```
>>> x, *y, *z = [[1, 2], [3, 4], 5];
>>> x, y, z
  File "<input>", line 1
Syntaxerror: two starred expressions in assignment
```

Використання індексів у списках

Список mylist				
№ елемента	1	2	3	4
Індекс	0	1	2	3

Оскільки нумерація елементів списку починається з 0, індекс останнього елемента буде на одиницю меншим від кількості елементів.

Функція `len()` – повертає кількість елементів списку

Приклад 3.80.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> len(arr) # Одержуємо кількість елементів
6
>>> arr[len(arr)-1] # Одержуємо останній елемент
    6
```

Некоректний індекс

Якщо елемент, відповідний до зазначеного індексу, відсутній у списку, то виконується виключення `IndexError`:

Приклад 3.81.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> arr[6]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range

>>>arr = [1, 'str', 4.1, '5']
>>> arr[5]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Від'ємний індекс

1. Як індекс можна вказати від'ємне значення.

2. Зсув буде відлічуватися від кінця списку.

Приклад 3.82.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> arr[-6], arr[-1] #перший і останній елемент
(1, 6)
```

Оскільки списки є змінюваними типами даних, то можна змінити елемент по індексу:

Приклад 3.83.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> arr[3] = 100 #змінюємо 4-й елемент
>>> arr
[1, 2, 3, 100, 5, 6]
>>> arr[-6] = 50 # Змінюємо (-6)-й елемент
```

```
>>> arr
[50, 2, 3, 100, 5, 6]
```

Операція зрізу

1. Списки підтримують операцію добування зрізу.

2. Операція повертає зазначений фрагмент списку.

Формат операції**:**

[<Початок>:<Кінець>:<Крок>]

Усі параметри не є обов'язковими.

1. Якщо параметр <Початок> не зазначений, то використовується значення 0.

2. Якщо параметр <Кінець> не зазначений, то повертається фрагмент до кінця списку. Слід також відмітити, що елемент з індексом, зазначеним в цьому параметрі, не входить у фрагмент, що повертається.

3. Якщо параметр <Крок> не зазначений, то використовується значення 1.

4. Як значення параметрів можна вказати від’ємні значення.

Застосування операції зрізу

Спочатку одержимо поверхневу копію списку:

Приклад 3.84.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> m = arr[1:6:2]
>>> #елемент з індексом 6 не входить у фрагмент, що повертається
>>> m
[2, 4, 6]
>>> m is arr
False
#якщо потрібно скопіювати всі елементи списку поверхнево
>>> s = arr[:]
>>> s
[1, 2, 3, 4, 5, 6]
>>> s is arr
False
```

Виведемо список у зворотному порядку.

Приклад 3.85.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> arr[::-1]   # крок -1
[6, 5, 4, 3, 2, 1]
```

Виведемо список без першого й останнього елементів.

```
>>> arr[1:] # Вивід списку без першого елемента
[2, 3, 4, 5, 6]
>>> arr[:-1]# Вивід без останнього елемента
[1, 2, 3, 4, 5]
>>> arr[0:2] # Одержимо перші два елементи
[1, 2] # Символ з індексом 2 не в діапазоні
>>> arr[-1:] # Останній елемент списку
[6]
>>> arr[1:4]# Повертаються елементи з індексами 1, 2 і 3
[2, 3, 4]
```

Зміна й видалення фрагмента списку

1. За допомогою зрізу можна змінити фрагмент списку.

2. Якщо зрізу присвоїти порожній список, то елементи, що потрапили в зріз, будуть вилучені:

Приклад 3.86.

```
arr = [1, 2, 3, 4, 5, 6]
arr[1:3] = [8,7]# Змінюємо елементи з індексами 1 і 2
print(arr)
результат: [1, 8, 7, 4, 5, 6]
arr[1:3]=[]#Видаляємо елементи з індексами 1 і 2
print(arr)
[ 1, 4, 5, 6]
```

```
arr = [1, 2, 3, 4, 5, 6]
arr[1]=8
arr[2]=7
print(arr)
```

Результат: [1, 8, 7, 4, 5, 6]

Конкатенація списків

З'єднати два списки в один список дозволяє оператор +. Результатом об'єднання буде новий список:

Приклад 3.87.

```
>>> arr1 = [1, 2, 3, 4, 5, 6]
>>> arr2 = [7, 8, 9]
>>> sum = arr1 + arr2
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Замість оператора + можна використовувати оператор +=. Слід враховувати, що в цьому випадку елементи додаються в поточний список:

Приклад 3.88.

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> arr += [7, 8, 9]
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Операції повторення (*) і перевірки на входження (in, not in) для списків

1. Повторити список зазначену кількість разів можна за допомогою оператора *.

2. Виконати перевірку на входження елемента в список дозволяє оператор `in`:

Приклад 3.89.

```
>>>arr= ["a", "b", "c"]*2 # Операція повторення
>>>arr
['a', 'b', 'c', 'a', 'b', 'c']
>>>arr[0] is arr[3]
```

```
True
>>> # Перевірка на входження
>>> "a" in ["a", "b", "c"], "d" in ["a", "b", "c"]
(True, False).
```

3.3.9. Багатовимірні списки

Будь-який елемент списку може містити об'єкт довільного типу.

Наприклад, елемент списку може бути:

- числом,

- рядком,

- списком,

- кортежем,

-словником і т. д.

Створити вкладений список можна, наприклад, так:

Приклад 3.90.

```
>>>nested = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> nested
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Вираз всередині дужок може розташовуватися на декількох рядках. Отже, попередній приклад можна записати інакше:

Приклад 3.91.

```
>>> nested = [                  #рівень1
              [1, 2, 3],        #рівень2
              [4, 5, 6],        #рівень2
              [7, 8, 9]         #рівень2
             ]                  #рівень1
>>> nested
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Щоб одержати значення елемента у вкладеному списку, слід указати два індекси:

Приклад 3.92.

```
>>> nested[0]
[1, 2, 3]
>>> nested[0][0]
1
```

Елементи вкладеного списку також можуть мати елементи довільного типу. Кількість вкладень не обмежена.

Можна створити об'єкт будь-якого ступеня складності.

Для доступу до елементів вказується кілька індексів підряд.

Приклад 3.93.

Елементи – списки.

```
>>> nested = [
              [["a", "b","A"],1],
              [["c", "d","B"],2],
              [["e", "f","C"],3]
            ]
>>> nested[1][1]
2
>>> nested[2][0][2]
'C'
```

Елементи – кортежі.

```
>>> nested = [
              [("a", "b","A"),1],
              [("c", "d","B"),2],
              [("e", "f","C"),3]
            ]
>>> nested[0][0][0], nested[1][0][1], nested[2][0][2]
('a', 'd', 'C')
```

Елементи – словники.

```
>>> nestdic = [1, "a", {"b":10,"c":["d", 100]}]
```

```
>>> nestdic[0]
1
>>> nestdic[1]
'a'
>>> nestdic[2]
{'c': ['d', 100], 'b': 10}
>>> nestdic[2]["c"]
['d', 100]
>>> nestdic[2]["c"][0]
'd'
```

3.3.10. Способи перебору елементів списку

Перебрати всі елементи списку можна за допомогою циклу `for`:

Приклад 3.94.

```
arr = [1, 2, 3, 4, 5, 6]
for i in arr: print(i, end=" ")
```

Результат:

1 2 3 4 5 6

```
arr = ["U","k", "r", "a", "i", "n", "e"]
for i in arr: print(i, end=" ")
```

Результат:

U k r a i n e

Зміна змінної циклу

1. Змінну **i** усередині циклу можна змінити.

2. Якщо змінна циклу посилається на незмінюваний тип даних (наприклад, число або рядок), то це не позначиться на початковому списку:

Приклад 3.95

```
# Елементи мають незмінюваний тип (число)

arr=[1, 2, 3, 4, 5]
for i in arr:
        i += 10 # i-type int
```

```
        print(i, end=' ')
print(arr)
```

Результат:

11 12 13 14 15 [1, 2, 3, 4, 5]

Список не змінився

Елементи мають змінюваний тип (список)

```
>>> arr = [[1, 2], [3, 4]]
>>> for i in arr: i[0] += 10 # i- type list
>>> arr # Список змінився
[ [11, 2], [13, 4]]
```

Функція `range()`

Функція `range()` має наступний формат:

`range``([<Початок>,]<Кінець>[,<Крок>])`

1. Перший параметр задає початкове значення. Якщо параметр <Початок> не зазначений, то за замовчуванням використовується значення 0.

2. У другому параметрі <Кінець> вказується кінцеве значення. Це значення не входить у діапазон значень, що повертається.

3. Якщо параметр <Крок> не зазначений, то використовується значення 1.

Функція **`range()`** використовується для одержання доступу до кожного елемента списку. Функція повертає об'єкт-діапазон, що підтримує ітерації, а за допомогою діапазону усередині циклу **for** можна одержати поточний індекс.

Помножимо кожний елемент списку на 2:

Приклад 3.96.

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print (arr)
# Результат виконання: [2, 4, 6, 8]
arr = [1, 2, 3, 4, 5, 6, 7, 8]
for i in range(len(arr)):
```

```
        if i>0: arr[i]+=arr[i-1]
print (arr)
```

Результат виконання: [1, 3, 6, 10, 15, 21, 28, 36]

Функція `enumerate`

Можна також скористатися функцією

`enumerate``(<Об'єкт>[,start=0]),`

Повертає кортеж з індексу й значення поточного елемента списку

<**`Об'єкт`**> – повинен підтримувати ітерації

[**`start`**] – початкове значення ітератора

Приклад 3.97.

```
>>> seasons=['Spring','Summer','Fall','Winter']
>>> list(enumerate(seasons))
[(0,'Spring'),(1,'Summer'),(2,'Fall'),(3,'Winter')]
>>> list(enumerate(seasons, start=1))
[(1,'Spring'),(2,'Summer'),(3,'Fall'),(4, 'Winter')]
```

Помножимо кожний елемент списку на 2:

Приклад 3.98.

```
arr = [1, 2, 3, 4]
for i, elem in enumerate(arr):
   elem *=2 # елемент arr не змінюється
   print(elem, end = " ")
print(arr)
for i, elem in enumerate(arr):
   arr[i] *= 2 # елемент arr змінюється
print (arr)
```

Результат:

2 4 6 8 [1, 2, 3, 4]

[2, 4, 6, 8]

Використання циклу `while`

1. Перебрати елементи можна за допомогою циклу `while`.

2. Слід пам'ятати, що цикл `while` працює повільніше від циклу `for`.

Помножимо кожний елемент списку на 2, використовуючи цикл `while`:

Приклад.

```
import random
arr = [1, 2, 3, 4]
arr1 = arr.copy()
while True:
    random.shuffle(arr1)
     if arr[3]- arr1[3]==3: break
print(arr1)
```

Результат:

[2, 3, 4, 1]

Генератори списків

У **прикладі** ми змінювали елементи списку таким чином:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
      arr[i] *= 2
print(arr) # Результат виконання: [2, 4, 6, 8]
```

Такий перебір можна зробити швидше і простіше, використавши генератор

```
arr_n = [i * 2 for i in arr]
```

Переваги використання генераторів:

1. За допомогою генераторів списків той же самий код можна записати більш компактно.

2. Генератори списків працюють швидше за цикл `for`.

Відмінність

3. Замість зміни початкового списку повертається новий список.

Генератор списку для перебору має формат:

<Новий список>=[<Інструкція> for ітератор in <Початковий список>]

1. **Інструкція**, виконувана у циклі, міститься перед циклом.

2. Інструкція в циклі **не містить оператора присвоювання**.

3. На кожній ітерації циклу буде генеруватися новий елемент, якому неявним чином присвоюється результат виконання виразу усередині циклу.

4. У підсумку буде створений новий список, що містить змінені значення елементів початкового списку.

Приклад 3.99. Використання генератора

```
arr = [ 1, 2, 3, 4]
arr_n = [i * 2 for i in arr]
print(arr_n) # Результат виконання: [2, 4, 6, 8]
```

Генератори списків зі складною структурою

1. Генератори можуть складатися з декількох вкладених циклів **for**.

2. Можуть містити оператор розгалуження **if** після циклу.

Одержимо парні елементи списку й помножимо їх на 10:

Приклад 3.100.

```
arr = [1, 2, 3, 4]
arr = [ i * 10 for i in arr if i % 2 == 0]
print(arr) # Результат виконання: [20, 40]
```

Приклад 3.101. Застосування циклу **for**:

```
arr = []
for i in [1, 2, 3, 4]:
    if i % 2 == 0:  # Якщо число парне
        arr.append(i*10) # Додаємо елемент
print(arr) # Результат виконання: [20, 40]
```

Одержимо парні елементи вкладеного списку й помножимо їх на 10:

Приклад 3.102.

```
arr = [[1, 2], [3, 4], [5, 6]]
arr1=[j * 10 for i in arr for j in i if j % 2 == 0]
print(arr1) # Результат виконання: [20, 40, 60]
```

Приклад 3.103. Застосування циклу **for**:

```
arr = [[1, 2], [3, 4], [5, 6]]
arr1 = []
```

```
for i in arr:
    for j in i:
    if j % 2 == 0: # Якщо число парне
            arr1.append(j * 10) # Додаємо елемент
print (arr1)
```

Результат виконання: [20, 40, 60]

3.4. Ітератори мови Python

3.4.1. Вирази-генератори

1. Якщо вираз розмістити усередині не в квадратних, а в круглих дужках, то буде повертатися не список, а ітератор.

2. Такі конструкції називають виразами-генераторами.

Приклад 3.104.

```
>>> arr = [1, 4, 12, 45, 10]
>>>j=(i for i in arr if i % 2 == 0)
>> next(j)
4
>>> next(j)
12
>>> next(j)
10
>>> next(j)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
>>>a=list(j)
>>>a
[4, 12, 10]
```

Застосування виразу генератора у функції `sum`

Приклад 3.105.

```
>>> arr = [1, 4, 12, 45, 10]
>>> j=(i for i in arr if i % 2 == 0)
>>> sum(j)
26
```

У прикладі використана функція

sum(<**елемент послідовності**>[, **start**])

Функція sum додає елементи списку зліва направо, починаючи з нульового елемента (за замовчуванням).

```
>>> arr = [1, 4, 12, 45, 10]
>>> sum(arr)
72
```

Застосування виразу генератора у циклі for

```
accum=0
arr = [1, 4, 12, 45, 10]
j=(i for i in arr if i % 2 == 0)
for k in j:
    accum+=k
print(accum)
```

Результат: 26

```
accum=0
for k in (i for i in [1, 4, 12, 45, 10] if i % 2 == 0):accum+=k
print(accum)
```

Результат: 26

3.4.2. Функція map()

Вбудована функція map() дозволяє застосувати функцію до кожного елемента послідовності. Функція має наступний формат:

map(<**посилання на функцію**>, <**Послідов1**>[, ... , <**ПослідовN**>])

1. Функція **map()** повертає ітератор

3. Параметр <**посилання на функцію**> містить посилання, у яку функцію буде передаватися поточний елемент послідовності.

Кількість параметрів даної функції повинна дорівнювати кількості послідовностей у параметрах функції `map`.

Додамо до кожного елемента **`elem`** списку число 10.

Приклад 3.106. Використаємо конструкцію: **ітератор**

```
def func(elem):
    """ Збільшення значення кожного елемента списку """
    return elem + 10 # Повертаємо нове значення
arr = [1, 2, 3, 4, 5]
j=map(func, arr)
for i in j: print(i, end=" ")
```

Результат: 11 12 13 14 15

Приклад 3.107. Використаємо функцію **`list()`**

```
def func(elem):
    """ Збільшення значення кожного елемента списку """
    return elem + 10 # Повертаємо нове значення
print(list(map(func, [1, 2, 3, 4, 5])), end=" ")
```

Результат: [11, 12, 13, 14, 15]

Приклад 3.108. Використаємо функцію **`tuple`**`()`

```
def func(elem):
    return elem * 10
arr = [1, 2, 3, 4, 5]
print(tuple(map(func, arr)), end=" ")
```

Результат: (10, 20, 30, 40, 50)

Приклад 3.109. Використаємо функцію **`set`**`()`

```
def func(elem):
    return elem - 10
arr = [1,2,2,2,2,2,2,2,3,4,5]
print(set(map(func, arr)), end=" ")
```

Результат:{-9, -8, -7, -6, -5}

Приклад 3.110. Використаємо функцію **sum**()

```
def func(elem):
    return elem**2
arr = [1, 2, 3, 4, 5]
J= map(func, arr)
print(sum(J))
```

Результат: 55

map *з декількома послідовностями*

Функції **map()** можна передати кілька послідовностей.

Для цього випадку функція повинна приймати стільки ж елементів, скільки маємо послідовностей у параметрах функції map().

У функцію зворотного виклику будуть передаватися одночасно елементи, які розташовані у послідовностях на однаковому зсуві.

```
def myf(k,m,n):
    return(k*m+n)
a1=[1,2,3]
a2=[4,5,6]
a3=[7,8,8]
j = map(myf,a1,a2,a3 )
print(list(j))
```

Результат: [11, 18, 26]

Приклад 3.111.

```
def func(e1, e2, e3):
    return e1 + e2 + e3 # Повертаємо нове значення
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
print(list(map(func, arr1, arr2, arr3)))
# Результат виконання: [111, 222, 333, 444, 555]
```

Приклад 3.112.

```
def dodan(e1, e2, e3):
    return e1 - e2 - e3
arr1 = [100, 200, 300, 400, 500]
arr2 = [10, 20, 30, 40, 50]
arr3 = [1, 2, 3, 4, 5]
print(list(map(dodan, arr1, arr2, arr3)))
# Результат виконання: [89, 178, 267, 356, 445]
```

Якщо кількість елементів у послідовностях буде різною, то як обмеження вибирається послідовність з мінімальною кількістю елементів:

Приклад 3.113.

```
def fdif(e1, e2, e3):
    """ Додавання елементів трьох різних списків """
   return e1 + e2 + e3
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20]
arr3 = [100, 200, 300, 400, 500]
print(list( map(fdif, arr1, arr2, arr3)))
# Результат виконання: [111, 222]
```

3.4.3. Вбудована функція `zip()`

Формат функції:

```
zip(<Послідовність1>[, ... ,<ПослідовністьN>])
```

1. На кожній ітерації повертає кортеж, що містить елементи послідовностей, які розташовані на однаковому зсуві.

Приклад 3.114.

```
>>> J=zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
>>> next(J)
(1, 4, 7)
>>> next(J)
```

```
(2, 5, 8)
>>> next(J)
(3, 6, 9)
>>> next(J)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Використання zip з функцією list()

Приклад 3.115.

```
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Приклад 3.116.

```
a = [1, 2, 3]
b = [4, 5, 6]
c = [7, 8, 9]
j = list(zip(a,b,c))
print("Список j =",j, end="")
s1,s2,s3 = sum(j[0]),sum(j[1]),sum(j[2])
print("\nСуми елементів з однаковим зсувом =",s1,s2,s3)
```

Результат:

Список **j** = [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

Суми елементів з однаковим зсувом = 12 15 18

Приклад 3.117.

```
J=zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
for i in J:
    print("{0}+{1}+{2}={3}".format(*i,sum(i)))
```

Результат:

1+4+7=12

2+5+8=15

3+6+9=18

Приклад 3.118.

```
J=zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
print(max(list(J)[0]))
```

Результат: 7

```
J=zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
print(list(J))
print(list(J))#!!!!!!!!
```

Результат

[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

[]

zip і кілька послідовностей різної довжини

Якщо кількість елементів у послідовностях буде різною, то в результат потраплять тільки елементи, які існують у всіх послідовностях на однаковому зсуві:

Приклад 3.119.

```
>>> list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10]))
```

[(1, 4, 7), (2, 6, 8)]

Приклад 3.120.

```
a=[1,2,3]
b=[4,6]
c=[7,8,9,10]
J = zip(a,b,c)
for i in J:
    print(i)
```

Результат:

(1, 4, 7)

(2, 6, 8)

Змінимо програму додавання елементів трьох списків з використанням функції `zip()` замість функції `map()`.

Приклад 3.121.

```
    arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
arr=[x + y + z for (x, y, z) in zip(arr1, arr2, arr3)]
print(arr)
# Результат виконання: [111, 222, 333, 444, 555]
arr=[sum(d) for d in zip(arr1, arr2, arr3)]
print(arr)
```

Приклад 3.122.

```
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400]
arr=[min(d) for d in zip(arr1,arr2,arr3)]
print(arr)# Результат:[1, 2, 3, 4]
```

3.4.4. Функція `filter()`

Функція **`filter()`** дозволяє виконати перевірку елементів послідовності.

Формат функції:

`filter`(<Функція>, <Послідовність>)

1. Якщо в першому параметрі замість назви функції вказати значення **`None`**, то кожний елемент послідовності буде перевірений на відповідність значенню **`True`**.

2. Якщо елемент у логічному контексті повертає значення `False`, то він не буде доданий в результат, що повертається.

3. Функція повертає ітератор – об'єкт, що підтримує ітерації.

4. Щоб одержати список, необхідно результат передати у функцію `list()`.

Приклад 3.123. Використання функції `filter`.

```
>>> J = filter(None, (1, 0, None, [], 2))
>>> next(J)
1
```

```
>>> next(J)
2
>>> next(J)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
>>> list(filter(None, [1, 0, None, [], 2]))
```

[1, 2]

Аналогічна операція з використанням генераторів списків має такий вигляд:

Приклад 3.124.

```
>>> [i for i in [1, 0, None, [], 2] if i]
```

[1, 2]

Filter + функція

У прикладі 30 замість параметра **None** можна вказати посилання на функцію.

У цю функцію як параметр буде передаватися поточний елемент послідовності.

Якщо елемент потрібно додати в значення,

що повертається функцією **filter()**,

то усередині функції зворотного виклику слід повернути значення **True**,

а якщо ні, то – значення **False**.

Приклад 3.125. Вилучимо всі від'ємні значення зі списку

Варіант з використанням циклу **for**

```
def fcheck(elem):
    return elem >= 0
arr = [-1, 2, -3, 4, 0, -20, 10]
J=filter(fcheck,arr)
for i in J:
    print(i, end=' ')
```

Варіант з використанням **list**

```
def func(elem):
    return elem >= 0
```

```
arr = [-1, 2, -3, 4, 0, -20, 10]
arr = list(filter(func, arr))
print(arr)  # Результат: [2, 4, 0, 10]
```

Приклад 3.126. Вилучити елементи списку з використанням генератора списків

```
def fcheck(elem):
    return elem >= 0
arr = [-1, 2, -3, 4, 0, -20, 10]
J=[i for i in filter(fcheck,arr)]
print([i for i in filter(fcheck,arr)])
```

Результат: [2, 4, 0, 10]

3.4.5. Функція `reduce()`

Функція **`reduce()`** з модуля `functools` застосовує зазначену функцію до пар елементів і накопичує результат.

Функція має наступний формат:

reduce(<Функція>, <Послідовність>[, <Початкове значення>])

У функцію зворотного виклику як параметри передаються два елементи:
перший елемент буде містити результат попередніх обчислень,
другий елемент – значення поточного елемента.

Приклад 3.127. Одержимо суму всіх елементів списку

```
import functools
def fred (x, y):
    print("({0}, {1})".format(x, y), end=" ")
    return x + y
arr = [1, 2, 3, 4, 5]
mysuma = functools.reduce(fred, arr,5)
print('варіант1',mysuma)
# Результат виконання:
```

```
(5, 1) (6, 2) (8, 3) (11, 4) (15, 5) варіант1 20
mysuma = functools.reduce(fred, arr)
print('варіант2',mysuma)
```

Результат виконання:

(1, 2) (3, 3) (6, 4) (10, 5) варіант2 15

```
mysuma = functools.reduce(fred, [], 10)
print('варіант3',mysuma)
# Результат виконання: варіант3 10
import functools
def fred (x, y):
    print("({0}, {1})".format(x, y), end=" ")
    return x*y
   arr = [1, 2, 3, 4, 5]
mysuma = functools.reduce(fred, arr)
print('варіант1',mysuma)
# Результат виконання:
    (1, 2) (2, 3) (6, 4) (24, 5) варіант1 120
mysuma = functools.reduce(fred, arr, 10)
print('варіант2',mysuma)
# Результат виконання:
(10, 1) (10, 2) (20, 3) (60, 4) (240, 5) варіант2 1200
mysuma = functools.reduce(fred, [1,2], 3)
print('варіант3',mysuma)
```

Результат виконання:

(3, 1) (3, 2) **варіант3 6**

```
from functools import *
arr1 = [5, 6, 7, 4, 45, 12]
arr2 = [2, 6, 1, 9, 10, 10]
def mfunc(a,b):
    if a>b: c=a-b
```

```
    else: c=0
    return c
def mred(x,y):
    return x+y
def mfil(k):
    if k%2==0: c=k
    else: c=0
    return c
print(reduce(mred,list(filter(mfil,[i**2 for i in
map(mfunc,arr1,arr2) if i < 32]))))
print([i**2 for i in map(mfunc,arr1,arr2) if i < 32])
print(list(filter(mfil,[i**2 for i in map(mfunc,arr1,arr2)
if i < 32])))
```

3.4.6. Додавання й видалення елементів списку

Для додавання й видалення елементів списку використовуються наступні методи:

`append (<Об'єкт>)` – додає один об'єкт у кінець списку. Метод змінює поточний список і нічого не повертає.

Приклад 3.128.

```
>>>arr = [1, 2, 3]
>>>arr.append(4) # Додаємо число
>>>arr
[1, 2, 3, 4]
>>> arr. append([5, 6])  # Додаємо список
>>> arr
[1, 2, 3, 4, [5, 6]]
>>> arr. append ( (7, 8) ) # Додаємо кортеж
>>> arr
[1, 2, 3, 4, [5, 6], (7, 8)]
```

extend(<Послідовність>) – додає елементи послідовності в кінець списку. Метод змінює поточний список і нічого не повертає.

Приклад 3.129.

```
>>> arr = [1, 2, 3]
```

```
>>> arr.extend([4, 5, 6]) # Додаємо список
>>> arr. extend ((7, 8, 9)) # Додаємо кортеж
>>> arr.extend("abc") # Додаємо букви з рядка
>>> arr
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 'a','b','c']
>>>arr.extend({"М","Н","О","Ж"})
>>>arr  # Додаємо множину
[1,2,3,4,5,6,7,8,9,'a','b','c','М','Ж','Н','О']
>>>arr.extend({"С":1,"Л":4,"О":3,"В":2})
>>>arr # Додаємо словник
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 'a" 'b', 'c', 'М', 'Ж', 'Н', 'О','С', 'Л', 'О', 'В']

Оператор += – додає елементи за допомогою операції конкатенації.

Приклад 3.130.

```
>>> arr = [1, 2, 3]
>>> arr + [4, 5, 6] # Повертає новий список
[1, 2, 3, 4, 5, 6]
>>> arr = [1, 2, 3]
>>> arr += [4, 5, 6]
>>> arr   # Змінює поточний список
```

[1, 2, 3, 4, 5, 6]

Крім того, можна скористатися операцією присвоювання значення зрізу:

Приклад 3.131. Використання зрізу

```
>>> arr = [1, 2, 3]
>>> arr[len(arr):] = [4, 5, 6]# Змінює список
>>> arr
```

[1, 2, 3, 4, 5, 6]

insert (<Індекс>, <Об'єкт>) – додає один об'єкт у зазначену позицію. Інші елементи зміщаються. Метод змінює поточний список і нічого не повертає.

Приклад 3.132.

```
>>> arr = [1, 2, 3]
>>> arr.insert(0, 0)
>>> arr # Вставляємо 0 у початок списку
[0, 1, 2, 3]
>>> arr.insert(-1, 20)
```

```
>>> arr # Можна вказати від’ємні
[0, 1, 2, 20, 3]
>>> arr.insert(2, 100)
>>> arr # Вставляємо 100 у позицію 2
[0, 1, 100, 2, 20, 3]
>>>arr.insert(10, [4, 5])
>>>arr
# Додаємо список
[0, 1, 100, 2, 20, 3, [4, 5]]
```

Метод **insert()** дозволяє додати тільки один об'єкт.

Щоб додати кілька об'єктів, можна скористатися операцією присвоювання значення зрізу. Додамо кілька елементів у початок списку:

Приклад 3.133.

```
>>> arr = [1, 2, 3]
>>> arr[:0] = [-2, -1, 0]
>>> arr
[-2, -1, 0, 1, 2, 3]
```

pop([<Індекс>]) – видаляє елемент, розташований по зазначеному індексу, і повертає його. Якщо індекс не зазначений, то видаляє й повертає останній елемент списку. Якщо елемента із зазначеним індексом немає, або список порожній, виконується виключення IndexError.

Приклад 3.134.

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.pop() # Видаляємо останній елемент списку
5
>>> arr # Список змінився
[1, 2, 3, 4]
>>> arr.pop(0) # Видаляємо перший елемент списку
1
>>>arr # Список змінився
```

```
[2, 3, 4]
```

Вилучити елемент списку дозволяє також оператор **del**:

Приклад 3.135.

```
>>>arr = [1, 2, 3, 4, 5]
>>>del arr[4] # Видаляємо останній елемент списку
>>>arr
```

[1, 2, 3, 4]

```
>>> del arr[:2] # Видаляємо перший і другий елементи
>>>arr
```

[3, 4]

```
>>> del arr # Видаляємо список
>>>arr
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'arr' is not defined
```

remove (<Значення>) – видаляє перший елемент, який має зазначене значення. Якщо елемент не знайдений, виконується виключення `ValueError`. Метод змінює поточний список і нічого не повертає.

Приклад 3.136.

```
>>> arr = [ 1, 2, 3, 1, 1 ]
>>> arr.remove(1) # Видаляє тільки першу одиничку
>>> arr
[2, 3, 1, 1]
>>> arr.remove(S) # Такого елемента немає
Traceback (most recent call last):
  File "<input>", line 1, in <module>
Nameerror: name 'S' is not defined
remove (<Значення>)
>>> arr=[2, 3, 1, 1]
>>> arr.remove(3) Видаляє елемент 3
```

```
>>> arr
[2, 1, 1]
>>> arr.remove(1) Видаляє елемент одиничку
>>> arr
[2, 1]
>>> arr.remove(5) # Такого елемента немає
Traceback (most recent call last):
  File "<input>", line 1, in <module>
Valueerror: list.remove(x): x not in list
```

clear() – видаляє всі елементи списку, очищаючи його. Жодного результату при цьому не повертається. Підтримка цього методу з'явилася в Python 3.

Приклад 3.137.

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.clear()
>>> arr
[]
```

Технологія вилучення повторень

Якщо необхідно вилучити всі повторювані елементи списку, то можна перетворити список у множину, а потім множину знов перетворити в список.

Список повинен містити тільки незмінювані об'єкти (наприклад, числа, рядки або кортежі). В протилежному випадку виконується виключення `TypeError`.

Приклад 3.138.

```
>>> arr = [ 1, 2, 3, 1, 1, 2, 2, 3, 3]
>>> s = set(arr) # Перетворимо список у множину
>>> s
{1, 2, 3}
>>> arr = list(s) # Перетворимо множину в список
>>> arr # Усі повтори були вилучені
[1, 2, 3]
```

3.5. Кортежі, множини та діапазони

Кортежі є незмінюваними типами даних. Іншими словами, можна одержати елемент по індексу, але змінити його не можна:

Приклад 3.139.

```
>>> t = (1, 2, 3) # Створюємо кортеж
>>> t[0] # Одержуємо елемент по індексу
1
>>> t[0] = 50 # Змінити елемент по індексу не можна!
TypeError: 'tuple' object does not support item assignment
```

Множини можуть бути як змінюваними, так і незмінюваними.

Їхня основна відмінність від щойно розглянутих типів даних: **1.** зберігання лише унікальних значень (неунікальні значення автоматично відкидаються). **2.** Елементи множини не індексуються.

Приклад 3.140. Створення множини

Створення зі списку:

```
>>> set([0, 1, 1, 2, 3, 3, 4])
{0, 1, 2, 3, 4}
>>> set(["string", 1, "string", 3, 3])
{'string', 1, 3}
```

Створення з кортежу:

```
>>> set(["a", "b", "c", "c", "d"])
{'a', 'd', 'b', 'c'}
```

Діапазон `range(start,end,step)`

Діапазони є наборами чисел, сформованими на основі заданих

- початкового значення величини (**`start`**),
- кінцевого значення (**`end`**),
- величини кроку між числами (**`step`**).

Найважливіша перевага перед усіма іншими наборами об'єктів – невеликий обсяг оперативної пам'яті для зберігання.

Приклад 3.141.

Якщо існує закономірність у послідовності

```
r = range(0, 101, 10)
for i in r: print(i, end = " ")
```

Результат роботи програми:

0 10 20 30 40 50 60 70 80 90 100

1. Щоб створити кортеж з одного елемента, необхідно наприкінці вказати кому

```
>>> t = (5,).
```

Саме коми формують кортеж, а не круглі дужки. Якщо усередині круглих дужок немає ком, то буде створений об'єкт іншого типу.

Приклад 3.142.

```
>>> t =(5)
>>> type(t) # Одержали число, а не кортеж!
<class 'int'>
>>> t = ("str")
>>> type(t) # Одержали рядок, а не кортеж!
<class 'str'>
>>> t =5,
>>> type(t)
<class 'tuple'>
```

Не дужки формують кортеж, а коми.

Будь-який вираз в мові Python можна взяти в круглі дужки.

Множина – це неупорядкована послідовність унікальних елементів, з якою можна порівнювати інші елементи, щоб визначити, чи належать вони цій множині.

Створити множину можна за допомогою функції **set**():

Приклад 3.143.

<table>
<tr><td>

```
>>>s=set(3,4)
>>> s
{3, 4}
```

</td><td>

```
>>>s=set([1,2])
>>>s
{1, 2}
```

</td><td>

```
>>>s=set("ab")
>>>s
{'b', 'a'}
```

</td></tr>
<tr><td colspan="2">

```
s = set({"a":1,"b":2})
s
{'b', 'a'}
```

</td><td>

```
>>> s = set()
>>> s
set()
```

</td></tr>
</table>

Функція `set()` дозволяє перетворити елементи послідовності в елементи множини.

3.5.1. Оператори й методи для роботи з множинами

Оператор | і метод **`union()`** – об'єднання двох множин:

Приклад 3.144.

```
>>> s = set([ 1, 2, 3])#створили множину зі списку
>>> a=s.union(set([4, 5, 6])),
>>> print(a)
{1, 2, 3, 4, 5, 6}
>>>s is a
False
>>> a=s | set([4, 5, 6])
>>> print(a)
{1, 2, 3, 4, 5, 6}
```

Оператор | і метод **`union()`** створюють новий об'єкт типу **`set`**

У об'єднання кожний елемент включається тільки один раз

Якщо даний елемент уже міститься в іншій множині, то він повторно доданий не буде.

Приклад 3.145.

```
a=set([1, 2, 3]) | set([1, 2, 3])
print("множина a:",a)
m=a.union(set([3, 4, 5]))
print("множина m:",m)
```

Результат:

```
множина a: {1, 2, 3}
множина m: {1, 2, 3, 4, 5}
```

Об'єднання множин без створення нового об'єкта

Оператор **a |= b** і метод **`a.update(b)`** – додають елементи множини `b` у множину `a`:

Приклад 3.146.

```
a=set([1, 2, 3, 4, 5, 6])
b=set([4, 5, 6, 7, 8, 9])
a.update(b)
print("множина a:",a)
```

Результат

множина a: {1, 2, 3, 4, 5, 6, 7, 8, 9}

Приклад 3.147.

```
a=set([1, 2, 3, 4, 5, 6])
b=set([4, 5, 6, 7, 8, 9])
a |= b
print("множина a:",a)
```

Результат:

множина a: {1, 2, 3, 4, 5, 6, 7, 8, 9}

Оператор - і метод difference()– обчислює різницю множин:

Приклад 3.148.

```
a=set([1, 2, 3, 6, 7])
b=set([1, 2, 4])
s=a-b
print("Оператор різниці:",s)
s=a.difference(b)
print("Метод різниці:",s)
```

Результат:

Оператор різниці: {3, 6, 7}

Метод різниці: {3, 6, 7}

Оператор - і метод **difference()** створюють новий об’єкт типу **set**

Різниця множин без створення нового об’єкта

Оператор **a -= b** і метод **a.difference_update(b)**

-= та **difference_update()** видаляють з множини a ті елементи, які одночасно існують в множині a і в множині b:

Приклад 3.149.

```
a=set([1, 2, 3, 6, 7])
b=set([1, 2, 4])
a-=b
print("Оператор:",a)
a.difference_update(b)
print("Метод:",a)
```

Результат:

Оператор: {3, 6, 7}

Метод: {3, 6, 7}

Оператор **&** і метод **intersection()** – виконують **перетин множин.**

В результаті одержуємо тільки ті елементи, які існують в обох множинах одночасно.

Приклад 3.150.

```
a=set([1, 2, 3, 6, 7])
b=set([1, 2, 4])
s=a & b
print("Оператор:",s)
s=a.intersection(b)
print("Метод:",s)
```

Результат:

Оператор: {1, 2}

Метод: {1, 2}

Перетин множин без створення нового об'єкта

Оператори **a &= b** і метод **a.intersection_update(b)**

У множині a залишаться елементи, які існують одночасно в множині a та множині b:

Приклад 3.151.

```
a=set([1, 2, 3, 6, 7])
b=set([1, 2, 4])
```

```
a &= b
print("Оператор:",a)
a=set([1, 2, 3, 6, 7])
a.intersection_update(b)
print("Метод:",a)
```

Результат:

Оператор: {1, 2}

Метод: {1, 2}

Оператор ^ і метод **symmetric_difference()** – повертають усі елементи обох множин, крім спільних елементів, які містяться в обох цих множинах:

Приклад 3.152.

```
a=set([1, 2, 3, 6, 7])
b=set([1, 2, 4])
s=a ^ b
print("Оператор:",s)
s=a.symmetric_difference(b)
print("Метод:",s)
```

Результат:

Оператор: {3, 4, 6, 7}

Метод: {3, 4, 6, 7}

Симетрична різниця без створення нового об'єкта

Оператори **a ^= b** і метод **a.symmetric_difference_update(b)**

У множині **a** будуть усі елементи обох множин, крім тих, що містяться в обох цих множинах:

Приклад 3.153.

```
a=set([1, 2, 3, 6, 7])
b=set([1, 2, 4])
a ^= b
print("Оператор:",a)
a=set([1, 2, 3, 6, 7])
```

```
a.symmetric_difference_update(b)
print("Метод:",a)
```

Результат:

Оператор: {3, 4, 6, 7}

Метод: {3, 4, 6, 7}

Оператори порівняння множин:

Оператор == – перевірка на рівність:

Приклад 3.154.

```
>>> set ( [1, 2, 3]) == set ( [1, 2, 3])
True
>>> set ( [1, 2, 3]) == set ( [3, 2, 1] )
True
>>> set ( [1, 2, 3]) == set ( [ 1, 2, 3, 4] )
False
>>> set(["a", "b", "c"]) == set(["a", "b", "c"])
True
>>> set(["a", "b", "c" ])=={"a", "b", "c", "d"}
False
>>> set(range(4))=={0,1,2,3}
True
res=set(range(4))==set(range(2,7))
print(res)
```

Результат роботи: False

Оператори **a <= b** і метод **a.issubset(b)** – перевіряють, чи входять усі елементи множини a в множину b.

Множина a може дорівнювати множині b.

Приклад 3.155.

```
>>> s = set ([1, 2, 3])
>>> s <= set([1, 2, 3])
True
```

```
>>> s <= set([1, 2]),
False
>>> s <= set([1, 2, 3, 4])
True
>>> s = set ([1, 2, 3, 4])
>>> s.issubset(set([1, 2]))
False
>>> s.issubset(set([1, 2, 3, 4, 5]))
True
```

Оператор **a < b** – перевіряє, чи строго входять усі елементи множини **a** в множину **b**, причому множина **a** не повинна дорівнювати множині **b**:

Приклад 3.156.

```
>>> s = set([ 1, 2, 3])
>>> s < set([1, 2, 3])
False
>>> s < set([1, 2, 3, 4])
True
>>> s = set (["a", "b", "c"])
>>> s < set(["a", "b", "c"])
False
>>> s < set(["a", "b", "c", "d"])
True
>>> set(range(3)) < set(range(4))
True
```

Оператори **a >= b** і метод **a.issuperset(b)** – перевіряють, чи не строго входять усі елементи множини **b** у множину **a**:

Приклад 3.157.

```
>>> s = set([1, 2, 3])
>>> s >= set([1, 2]), s >= set([1, 2, 3, 4])
(True, False)
```

```
>>> s.issuperset(set([1,2]))
True
>>> s.issuperset(set([1, 2, 3, 4]))
False
>>> s = set (["a", "b", "c"])
>>> s.issuperset(set(["a", "b"]))
True
>>> s >= set(["a", "b"])
True
```

Оператор **a > b** – перевіряє, чи входять усі елементи множини **b** у множину **a**, причому множина **a** не повинна дорівнювати множині **b**:

Приклад 3.158.

```
>>> s = set([1, 2, 3])
>>> s > set([1, 2])
True
>>> s > set([1, 2, 3])
False
>>> s > set([1, 2, 3, 4])
False
>>> s = set (["a", "b", "c"])
>>> s > set(["a", "b", "c"])
False
>>> s > set(["a", "b"])
True
```

Метод **a.isdisjoint(b)** – перевіряє, чи є множини **a** й **b** повністю різними, тобто не утримуючими жодного співпадаючого елемента:

Приклад 3.159.

```
>>> s = set([1, 2, 3])
>>> s.isdisjoint(set([4, 5, 6]))
True
```

```
>>> s.isdisjoint(set([1, 3, 5]))
False
>>> s = set (["a", "b", "c"])
>>> s.isdisjoint(set(["a", "b"]))
False
>>> s.isdisjoint(set(["d", "e"]))
True
```

3.5.2. Інші методи для роботи з множинами

Метод **copy**() – створює копію множини.

Примітка!!!!: оператор = присвоює лише посилання на той же об'єкт, а не копіює його.

Приклад 3.160.

```
>>> s = set([1, 2, 3])
>>> c=s; s is c # За допомогою = копію створити не можна!
True
>>> c = s.copy() # Створюємо копію об'єкта
>>> c
{1, 2, 3}
>>> s is c       # Тепер це різні об'єкти
False
```

Метод **add (<Елемент>)** – додає <Елемент> у множину:

Приклад 3.161.

```
>>> s = set([1, 2, 3])
>>> s.add(4); s
{1, 2, 3, 4}
my = {"Petrenko"}
my.add("Ivan")
print (my)
```

```
my.add("Ivanovich")
print (my)
```

Результат:

{'Ivan', 'Petrenko'}

{'Ivanovich', 'Ivan', 'Petrenko'}

Метод **remove(<Елемент>)** – видаляє <Елемент> із множини. Якщо елемент не знайдений, то виконується виключення **KeyError**:

Приклад 3.162.

```
>>> s = set ([1, 2, 3] )
>>> s.remove(3); s # Елемент існує
{1, 2}
>>> s.remove(5) # Елемент НЕ існує
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 5
my = {"Petrenko", "Sydorenko"}
my.remove("Sydorenko")
print(my)
```

Результат: {'**Petrenko**'}

Метод **discard (<Елемент>)** – видаляє <Елемент> із множини, якщо він присутній. Якщо зазначений елемент не існує, ніякого виключення не виконується:

Приклад 3.163.

```
>>> s = set([1, 2, 3])
>>> s.discard(3); s # Елемент існує
{1, 2}
>>> s.discard(5); s # Елемент НЕ існує
{1, 2}
my = {"Petrenko", "Sydorenko"}
print(my)
my. discard ("Ivanov")
```

```
print(my)
```

Результат:

{'Petrenko', 'Sydorenko'}

{'Petrenko', 'Sydorenko'}

Метод **pop()** – видаляє елемент із множини й повертає його. Якщо елементів немає, то виконується виключення `KeyError`. Увага! Тип «set» не підтримує індексацію елементів!!!! Тому метод **pop** використовується без параметрів.

Приклад 3.164.

```
>>> s = set([1, 2])
>>> s.pop()
1
>>> s
{2}
>>> s.pop()
2
>>> s
set()
>>> s.pop() # Якщо немає елементів, то помилка
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Метод **clear()** – видаляє всі елементи із множини:

```
>>> s = set([1, 2, 3])
>>> s.clear()
>>> s
set()
my = {"Petrenko", "Sydorenko"}
my.clear()
print("Множина:",my, "Довжина множини:",len(my))
```

Результат: Множина: `set()` Довжина множини: 0

3.5.3. Генератори множин

1. Синтаксис генераторів множин схожий на синтаксис генераторів списків.

2. Відмінність у тому, що вираз міститься у фігурних дужках, а не у квадратних.

Розглянемо приклад, де результатом є множина, у якій всі повторювані елементи будуть вилучені.

Приклад 3.165.

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3]}
{1, 2, 3}
>>> {x for x in ["a", "b", "b", "a", "c"]}
{'b', 'c', 'a'}
>>> {x for x in ["winter", "summer", "fall", "fall",
"spring"]}
{'summer', 'fall', 'spring', 'winter'}
```

Генератори множин зі складною структурою

1. Генератори множин можуть складатися з декількох вкладених циклів **for**.

2. Можуть містити оператор розгалуження **if** після циклу.

Приклад 3.166. Унікальні парні елементи.

```
>>> {x for x in [1,2,1,2,1,2,3] if x % 2 == 0}
{2}
>>> {x for x in [1,2,1,2,1,2,3] if x < 3}
{1, 2}
>>> {x for x in [1,2,1,2,1,2,3] if (x<3)&(x>1)}
{2}
>>> {x for x in list(zip([1,2],[1,2],[1,2,3]))
if x[0] % 2 == 0}
{(2, 2, 2)}
```

Тип множин **frozenset**. На відміну від типу **set**, множину типу **frozenset** не можна змінити.

Оголосити множину можна за допомогою функції `frozenset()`:

Приклад 3.167.

```
>>> f = frozenset()
>>> f
frozenset()
```

Функція `frozenset()` дозволяє також перетворити елементи послідовності в множину:

Приклад 3.168.

```
>>> frozenset("string") # Перетворимо рядок
frozenset({'i', 'r', 'g', 's', 'n', 't'})
>>> frozenset([1, 2, 3, 4, 4]) # Перетворимо список
frozenset({1, 2, 3, 4})
>>> frozenset((1, 2, 3, 4, 4))# Перетворимо кортеж
frozenset({1, 2, 3, 4})
```

Діапазони

range([<Початок>,] <Кінець> [, <Крок>])

Перетворити діапазон у список, кортеж, звичайну або незмінювану множину можна за допомогою функцій `list()`, `tuple()`, `set()` або `frozenset()` відповідно:

Приклад 3.169.

```
>>> list(range(1, 10)) # Перетворимо в список
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(1, 10)) # Перетворимо в кортеж
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> set(range(1, 10)) # Перетворимо в множину
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> frozenset(range(1, 10))
>>> frozenset({1, 2, 3, 4, 5, 6, 7, 8, 9})
```

3.5.4. Функції, методи та оператори для `range()`

Приклад 3.170.

```
# застосування функції len()
>>>len(range(20,3,-2))
9
```

```
# застосування функції min()
>>> min(range(20,3,-2))
4
# застосування функції max()
>>> max(range(0,127,3))
126
# метод index()
>>> range(0,127,3).index(15)
5
- метод count()
>>> range(0,127,3).count(16)
0
```

Оператори порівняння діапазонів

Оператор == – повертає **True**, якщо діапазони рівні, і **False** якщо ні.

Діапазони вважаються рівними, якщо вони містять однакові послідовності чисел!!

Приклад 3.171.

```
>>> range(1, 10) == range(1, 10, 1)
True
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 10,1))
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(1, 10, 2) == range(1, 12, 2)
False
>>> list(range(1, 10,2))
[1, 3, 5, 7, 9]
>>> list(range(1, 12,2))
[1, 3, 5, 7, 9, 11]
```

Оператор != – повертає **`True`**, якщо діапазони не рівні, і **`False`** в протилежному випадку:

Приклад 3.172.

```
>>> range(1, 10, 2) != range(1, 12, 2)
True
>>> list(range(1, 10,2))
[1, 3, 5, 7, 9]
>>> list(range(1, 12,2))
[1, 3, 5, 7, 9, 11]
>>> range(1, 10) != range(1, 10, 1)
False
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 10,1))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Атрибути `start, stop і step` повертають, відповідно,

`start` – початкову границю діапазону,

`stop` – кінцеву границю діапазону,

`step` – крок діапазону.

Приклад 3.173.

```
>>> r = range(1, 10)
>>> r.start
1
>>>r.stop
10
>>>r.step
1
>>> r = ange(1, 11, 2)
>>> r.start, r.stop, r.step
(1, 11, 2)
```

3.6. Словники. Основні характеристики словників

Словники – це набори об'єктів, доступ до яких здійснюється не по індексу, а по ключу.

1. Кожний елемент словника містить пари:

<**ключ**>:<**значення**>

2. Ключ – це незмінюваний об'єкт:

число, рядок або кортеж.

3. Елементи словника можуть:

містити об'єкти довільного типу даних,

мати необмежений ступінь вкладеності.

4. Елементи в словниках розташовуються в довільному порядку.

5. Щоб одержати елемент, необхідно вказати ключ, який використовувався при збереженні значення.

6. Словники є відображеннями, а не послідовностями.

Тому функції списків і кортежів до них незастосовні.

7. Як і списки, словники є змінюваними типами даних. Іншими словами, ми можемо не тільки одержати значення по ключу, але й змінити його.

3.6.1. Створення словника

Існує 4 способи створення словників.

Спосіб 1. Створюємо словник за допомогою функції **dict()** .

Формати функції:

dict(<Ключ1>=<Значення1>[,...,<КлючN>=<ЗначенняN>])

dict(<Словник>)

dict(<Список кортежів з двома елементами (Ключ, Значення)>)

dict(<Список списків з двома елементами [Ключ, Значення]>)

Якщо параметри не зазначені, то створюється порожній словник.

Приклад 3.174. Створення словників у **спосіб 1**

```
>>> d = dict()  # Створюємо порожній словник
>>> d
```

```
{ }
>>> d = dict(a = 1, b = 2)#ключ=значення
>>> d
{'b': 2, 'a': 1}
>>> d = dict({"a" : 1, "b" :  2})#Словник
>>> d
{'b': 2, 'a': 1}
# Список кортежів
>>> d = dict ([("a",1), ("b", 2)])
>>> d
{'b': 2, 'a': 1}
# Список списків
>>> d = dict ([["a",1], ["b", 2]])
>>> d
{'b': 2, 'a': 1}
```

Створення словників за способом 1 у циклі

Приклад 3.175.

```
from random import sample
b='abcdefghijklmnopqrstuvwxyz'
def fn(a):
   c="".join(sample(b,4))
      return (a,c)
d= map(fn,range(5))
   print(dict(d))
```

Результат

```
{0:'jkyz',1:'uiha',2:'vcas',3:'jwvs',4:'qnpt'}
```

Використання функції `zip()`

1. Об’єднуєм два списки в список кортежів за допомогою функцій **`list`**`()` і **`zip`**`()`.

2. Потім цей список можна використовувати для створення словника.

Приклад 3.176.

```
>>> k = ["a", "b"] # Список з ключами
>>> v = [1, 2]     # Список зі значеннями
>>> a=list(zip(k, v))# [('a', 1), ('b', 2)]
>>> d = dict (a)
>>> d
{'a': 1, 'b': 2}
>>> d = dict(zip(k, v))
>>> d # Створення словника
{'a': 1, 'b': 2}
```

Спосіб 2. Створюємо словник, вказавши всі елементи всередині фігурних дужок.

Це найбільш часто використовуваний спосіб створення словника. Між ключем і значенням вказують двокрапку, а пари «ключ/значення» записують через кому.

Приклад 3.177.

```
>>> d = {}
>>> d # Створення порожнього словника
{}
>>> d = {"Ann":19, "Wolf":22 }
>>> d
{'Ann': 19, 'Wolf': 22}
>>> d = {"Пн":1, "Вт":2, "Ср":2, "Чт":2}; d
{'Вт': 2, 'Ср': 2, 'Пн': 1, 'Чт': 2}
>>> d = {("Пн","Вер"):1, ("Вт","Жовт"):2}; d
{('Пн', 'Вер'): 1, ('Вт', 'Жовт'): 2}
```

Спосіб 3. Створюємо словник, заповнивши словник поелементно.

У цьому випадку ключ вказується усередині квадратних дужок:

Приклад 3.178.

```
>>> d = {}     # Створюємо порожній словник
>>> d["a"] = 1 # Додаємо елемент1 (ключ "a")
>>> d["b"] = 2 # Додаємо елемент2 (ключ "b")
```

```
>>> d["c"] = 3 # Додаємо елемент3 (ключ "c")
>>> d
{'b': 2, 'a': 1, 'c': 3}
>>> nest = {}
>>> nest["Africa","South"]=1
>>> nest["America","West"]=2
>>> nest
{('America', 'West'): 2, ('Africa', 'South'): 1}
```

Спосіб 4. Створюємо словник за допомогою методу

dict.fromkeys(<Послідовність>[, <Значення>])

Метод створює новий словник.

1. Ключами словника будуть елементи послідовності, переданої першим параметром.

2. Значенням елементів словника буде величина, передана другим параметром.

3. Якщо другий параметр не зазначений, то значенням елементів словника буде значення **None**.

Приклад 3.179.

```
# немає другого параметра
>>> d=dict.fromkeys(["winter","summer","spring"])
>>> d
{'winter': None, 'spring': None, 'summer': None}
# Ключ – кортеж, другий параметр 0
>>> d = dict.fromkeys(("Пн", "Вт", "Ср"), 0)
>>> d
{'Ср': 0, 'Пн': 0, 'Вт': 0}
```

Групове присвоювання дає одне і те ж посилання на об’єкт

1. При створенні словника в змінній зберігається посилання на об'єкт, а не сам об'єкт. Це обов'язково слід враховувати при груповому присвоюванні.

2. Групове присвоювання можна використовувати для чисел і рядків, але для списків і словників цього робити не можна.

Приклад 3.180.

```
# Нібито створили два об'єкти
>>> d1 = d2 = { "a": 1, "b": 2}
>>> d2["b"] = 10
>>> d1, d2 # Змінилося значення у двох змінних
({'a': 1, 'b': 10}, {'a': 1, 'b': 10})
```

Як видно з прикладу, зміна значення в змінній **d2** привела також до зміни значення в змінній **d1**. Тобто, обидві змінні посилаються на той самий об'єкт, а не на два різних об'єкти.

Позиційне присвоювання для словників

Щоб одержати два об'єкти, виконуємо позиційне присвоювання.

Приклад 3.181.

```
>>> d1, d2 = {"a": 1, "b": 2}, {"a": 1, "b": 2}
>>> d2["b"] = 10
>>> d1, d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
group1={"John": 23, "Mary": 18}
group2={"John": 23, "Mary": 18}
group1["John"]=40
print(group1,group2)
```

Результат роботи:

```
{'John': 40, 'Mary': 18} {'John': 23, 'Mary': 18}
```

Створити поверхневу копію словника функцією `dict()`

Формат: `<Словник2> = dict(<Словник1>)`

Приклад 3.182.

```
>>> d1 = {"a": 1, "b": 2} # Створюємо словник
>>> d2 = dict(d1) # Створюємо поверхневу копию
>>> d1 is d2 # це різні об'єкти
False
>>> d2["b"] = 10
```

```
>>> d1, d2 # Змінилася тільки змінна d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

3.6.2. Створення поверзневої та глибокої копії словника

Створити поверхневу копію словника методом `copy()`

Приклад 3.183.

```
>>> d1 = {"a": 1, "b": 2} # Створюємо словник
>>> d2 = d1.copy() # Створюємо поверхневу копію
>>> d1 is d2 # Оператор показує, що це різні об'єкти
False
>>> d2 ["b"] = 10
>>> d1, d2 # Змінилося тільки значення в змінній d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Створити поверхневу копію дворівневого словника

Приклад 3.184.

```
>>> d1 = {"a": 1, "b": [20, 30, 40]}
>>> d2 = dict(d1) # Створюємо поверхневу копію
>>> d2 ["b"][0] = "test"
>>> d1 # Змінилися дві змінні
{'a': 1, 'b': ['test', 30, 40]}
>>> d2 # Змінилися дві змінні
{'a': 1, 'b': ['test', 30, 40]})
```

Створити глибоку копію словника функцією `deepcopy()` *з модуля* `copy`

Приклад 3.185.

```
>>> import copy
>>> d3 = copy.deepcopy(d1)#створюємо глибоку копію
>>> d3["b"][1] = 800
>>> d1
({'a': 1, 'b': ['test', 30, 40]},
>>> d3 # Змінилося значення тільки в змінній
{'a': 1, 'b': ['test', 800, 40]})
```

3.6.3. Доступ до елементів словника

1. Доступ до елементів словника здійснюється за допомогою квадратних дужок, у яких вказується ключ.

2. Як ключ можна вказати незмінюваний об'єкт: число, рядок або кортеж.

Приклад 3.186.

```
>>> d = {1: "int", "a": "str", (1, 2): "tuple"}
>>> d[1]
'int'
>>> d["a"]
'str'
>>> d[(1, 2)]
'tuple'
```

Доступ до неіснуючого елемента словника

Якщо елемент, відповідний до зазначеного ключа, відсутній у словнику, то виконується виключення **KeyError**:

Приклад 3.187.

```
>>> d = {"Пн": 1, "Вт": 2}
>>> d["Ср"]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'Ср'
# Доступ до неіснуючого елемента
```

Перевірка існування ключа за допомогою операторів `in` *і* `not in`

Оператор **in**. Якщо ключ знайдений, то повертається значення **True**, а якщо ні, то – **False**.

Приклад 3.188.

```
>>> d = { "a": 1, "b": 2}
>>> "a" in d # Ключ існує
True
>>> "c" in d # Ключ не існує
False
```

Оператор **not in**. Якщо ключ відсутній, повертається **True**, інакше – **False**.

Приклад 3.189.

```
>>> d = {"a": 1, "b": 2}
>>> "c" not in d # Ключ не існує
True
>>> "a" not in d # Ключ існує
False
```

Метод get

Формат методу: **get**(<Ключ>[, <Значення за замовчуванням>])

1. За відсутності ключа повертається **None**, якщо другий параметр не заданий,

2. За відсутності ключа повертається значення другого параметра, якщо воно задане.

3. Якщо ключ присутній у словнику, то метод повертає значення, відповідне до цього ключа.

Приклад 3.190.

```
d = {"a": 1, "b": 2}
m=d.get("a")
print(m)
```

Результат: 1

```
n=d.get("a",200)
print(n)
```

Результат: 1

```
a=d.get("c")
print(a)
```

Результат: None

```
a=d.get("c", 800)
print(a)
```

Результат: 800

```
d.get()
TypeError: get expected at least 1 arguments, got 0
```

Метод `setdefault`

Формат методу: **`setdefault(`**<Ключ>[, <Значення за замовчуванням>])

1. Якщо ключ відсутній, то:

- у словнику створюється новий елемент зі значенням, зазначеним у другому параметрі;
- якщо другий параметр не зазначений, значенням нового елемента буде `None`.

2. Якщо ключ присутній у словнику, то метод повертає значення, відповідне до цього ключа.

Приклад 3.191.

```
d = {"a": 1, "b": 2}
a=d.setdefault("a")
print("a={0!s};  d={1!s}".format(a,d))
Результат: a=1;  d={'a': 1, 'b': 2}
a=d.setdefault("a",0)
print("a={0!s};  d={1!s}".format(a,d))
Результат: a=1;  d={'b': 2, 'a': 1}
a=d.setdefault("c")
print("a={0!s};  d={1!s}".format(a,d))
Результат:a=None;  d={'a': 1, 'c': None, 'b': 2}
a=d.setdefault("c",0)
print("a={0!s};  d={1!s}".format(a,d))
Результат: a=0;  d={'c': 0, 'b': 2, 'a': 1}
```

3.6.4. Модифікація словника

1. Словники є змінюваними типами даних.

2. Можна змінювати елемент по ключу.

3. Якщо елемент із зазначеним ключем відсутній у словнику, то він буде доданий у словник.

Приклад 3.192.

```
>>> d = {"a": 1, "b": 2}
>>> d ["a"] = 800  # Зміна елемента по ключу
```

```
>>> d["c"] = "string" # Буде доданий елемент
>>> d
{'b': 2, 'a': 800, 'c': 'string'}
```

Визначення довжини словника

Одержати кількість ключів у словнику дозволяє функція **len()**:

Приклад 3.193.

```
>>> d = {"a": 1, "b": 2}
>>> len(d) #Одержуємо кількість ключів у словнику
2
```

Вилучення елемента словника

Вилучити елемент із словника можна за допомогою оператора **del**:

Приклад 3.194.

```
d = { "a": 1, "b": 2}
del d["b"]
print(d) # Видаляємо елемент з ключем "b"
{'a': 1}
```

3.6.5. Перебір елементів словника

Перебрати всі елементи словника можна за допомогою циклу **for**, хоча словники й не є послідовностями.

Два способи виводу елементів словника:

1. Перший спосіб використовує метод **keys()**, що повертає об'єкт з ключами словника.

2. У другому випадку ми просто вказуємо словник як параметр.

3. На кожній ітерації циклу буде **повертатися ключ**, за допомогою якого усередині циклу можна одержати значення, що відповідає цьому ключу.

Приклад 3.195.

```
#спосіб 1
d = {"x": 1, "y": 2, "z": 3}
for key in d.keys(): # Використання методу keys()
    print("{0}:{1}".format(key, d[key]), end=" ")
```

Результат роботи: y:2 z:3 x:1

```
#спосіб 2
d = {"x": 1, "y": 2, "z": 3}
for key in d: # Словники також підтримують ітерації
    print("{0!s}: {1!s},".format(key, d[key]), end=" ")
```

Результат роботи: z:3, x:1, y:2,

Сортування по ключах методом `sort()`

1. Словники є неупорядкованими структурами. Тому елементи словника виводяться в довільному порядку.

2. Щоб вивести елементи з сортуванням по ключах, слід одержати список ключів, а потім скористатися методом **`sort()`**.

Приклад 3.196.

```
d = {"x": 1, "y": 2, "z": 3}
k = list(d.keys())  # Одержуємо список ключів
k. sort()           # Сортуємо список ключів
for key in k:
  print("({0!s} => {1!s})".format(key, d[key]), end=" ")
```

Результат роботи: (x => 1) (y => 2) (z => 3)

Сортування по ключах функцією `sorted()`

Приклад 3.197.

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d.keys()):
   print("({0!s} => {1!s})".format(key,d[key]),end=" ")
```

Результат роботи: (x => 1) (y => 2) (z => 3)

Оскільки на кожній ітерації повертається ключ словника, функції `sorted()` можна відразу передати об'єкт словника, а не результат виконання методу `keys()`.

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d):
    print("({0!s} => {1!s})".format(key,d[key]),end=" ")
```

Результат роботи: (x => 1) (y => 2) (z => 3)

Методи для роботи зі словниками (робота з ключами)

Метод keys() – повертає об'єкт dict_keys, що містить усі ключі словника. Цей об'єкт підтримує ітерації, а також операції над множинами.

Приклад 3.198.

```
# Одержуємо об'єкт dict keys
>>> d1 = {"a": 1, "b": 2}
>>> d2 = {"a": 3, "c": 4, "d": 5}
>>> d1.keys(), d2.keys()
(dict_keys(['a', 'b']),dict_keys(['d', 'a', 'c']))
# Одержуємо кортеж списків ключів
>>> list(d1.keys()), list(d2.keys())
(['a', 'b'], ['d', 'a', 'c'])
# Перебір ключів
>>> for k in d1.keys(): print(k, end=" ")
a b
# Множина об’єднання ключів
d1, d2 = {"a": 1, "b": 2}, {"a": 3, "c": 4, "d": 5}
A = d1.keys() | d2.keys()
print(A)
```

Результат роботи: {'c', 'd', 'a', 'b'}

```
# Множина різниці ключів
A = d1.keys() - d2.keys()
print(A)
```

Результат роботи:{'b'}

```
# Множина однакових ключів
A = d1.keys() & d2.keys()
print(A)
```

Результат роботи: {'a'}

```
# Множина унікальних ключів
>>> A = d1.keys() ^ d2.keys()
```

```
{'d', 'b', 'c'}
```

Методи для роботи зі словниками (значення)

Метод values() – повертає об'єкт dict_values, що містить усі значення словника.

Цей об'єкт підтримує ітерації.

Приклад 3.199.

```
>>> d = {"a": 1, "b": 2}
>>> d.values()  # Одержуємо об'єкт dict values
dict_values([1, 2])
>>> list(d.values()) # Одержуємо список значень
[1, 2]
>>> [ v for v in d.values()]#Генератор значень
[1, 2]
```

Методи для роботи зі словниками (елементи)

Метод items() – повертає об'єкт dict_items, що містить усі ключі й значення у вигляді кортежів. Цей об'єкт підтримує ітерації.

Приклад 3.200.

```
>>> d = {"a": 1, "b": 2}
>>> d.items() # Одержуємо об'єкт dict items
dict_items([('a', 1), ('b', 2)])
>>> list(d.items())# Одержуємо список кортежів
[('a', 1), ('b', 2)]
```

Метод pop(<Ключ>*[, <Значення за замовчуванням>])*

pop(<Ключ>) видаляє елемент із зазначеним ключем і повертає його значення.

pop(<Ключ>, (Значення)) якщо ключ відсутній, то повертається значення із другого параметра.

pop(<Ключ>) ключ відсутній, і другий параметр не зазначений, виконується виключення **KeyError.**

Приклад 3.201.

```
d = {"a": 1, "b": 2, "c": 3}
m = d.pop("a")
print("m = {0!s};  d = {1!s}".format(m,d))
```

Результат: **m = 1;** d = {'b': 2, 'c': 3}

```
d = {"a": 1, "b": 2, "c": 3}
m = d.pop("a",245)
print("m = {0!s};  d = {1!s}".format(m,d))
```

Результат: **m = 1;** d = {'b': 2, 'c': 3}

```
m = d.pop("n",245)
print("m = {0!s};  d = {1!s}".format(m,d))
```

Результат: **m = 245;**d = {'c': 3, 'b': 2, 'a': 1}

```
m = d.pop("n")
print("m = {0!s};  d = {1!s}".format(m,d))
```

KeyError: 'n'

Метод popitem()

Цей метод видаляє довільний елемент і повертає кортеж із ключа й значення.

Якщо словник порожній, виконується виключення **KeyError**.

Приклад 3.202.

```
>>> d = {"a": 1, "b": 2}
>>> d.popitem() # Видаляємо довільний елемент
('a', 1)
>>> d.popitem() # Видаляємо довільний елемент
('b', 2)
>>> d.popitem()#Словник порожній. Виконується виключення
Traceback (most recent call last):
  File "<input>", line 1, in <module>
```

```
KeyError: 'popitem(): dictionary is empty'
```

Метод `clear()`

Цей метод видаляє всі елементи словника. Метод нічого не повертає як значення.

Приклад 3.203.

```
>>> d = {"a": 1, "b": 2}
>>> d.clear() # Видаляємо всі елементи
>>> d         # Словник тепер порожній
{ }
```

`Метод update ()` – додає елементи в словник.

Метод змінює поточний словник і нічого не повертає.

Формати методу:

`update`(<Ключ1>=<Значення1>[, ... , <Ключn>=<Значеннsn>])

`update`(<Словник>)

`update`(<Список кортежів з двома елементами>)

`update`(<Список списків з двома елементами>)

Якщо елемент із зазначеним ключем уже присутній у словнику, то його значення буде перезаписано.

Приклад 3.204.

```
d = {"a": 1, "b": 2}
d.update(c=3, d=4, f=123)
print(d)
{'f': 123, 'c': 3, 'b': 2, 'd': 4, 'a': 1}
d.update({"c": 10, "d": 20}) # Словник
print(d)# Значення елементів перезаписані
{'d': 20, 'b': 2, 'a': 1, 'f': 123, 'c': 10}
d.update([("d", 80), ("e", 6)])#Список кортежів
print(d)
{'d': 80, 'e': 6, 'b': 2, 'a': 1, 'f': 123, 'c': 10}
d.update([["a","str"],["i","t"]])#Список списків
```

```
print(d)
{'d':80,'e':6,'b':2,'i':'t','a':'str','f':123,'c':10}
```

Генератори словників

Синтаксис генераторів словників схожий на синтаксис генераторів списків, але має дві відмінності:

1. Вираз міститься у фігурних дужках {…}, а не у квадратних.

2. Усередині виразу перед циклом **for** вказуються два значення через двокрапку, а не одне `{a:b for….}`

Значення, розташоване ліворуч від двокрапки, стає ключем.

Значення, розташоване праворуч від двокрапки, – значенням елемента.

Простий генератор словників

Приклад 3.205.

```
# Список з ключами
>>> mykeys = ["Anna", "Maria","Bonni"]
# Список зі значеннями
>>> values = [18, 21, 50]
>>> {k:v for (k, v) in zip(mykeys, values)}
{'Maria': 21, 'Bonni': 50, 'Anna': 18}
>>> {k:10 for k in mykeys}
{'Maria': 10, 'Bonni': 10, 'Anna': 10}
```

Генератор словників із складною структурою

Генератори словників можуть мати складну структуру. Вони можуть складатися з декількох вкладених циклів **for** і (або) містити оператор розгалуження **if** після циклу.

Створимо новий словник, що містить тільки елементи з парними значеннями, з початкового словника:

Приклад 3.206.

```
d={"Maxim":11, "Bart":12, "Dago":15, "Whit": 18}
p={k: v for (k,v) in d.items() if v % 2 == 0}
print(p)
```

Результат: `{'Bart': 12, 'Whit': 18}`

Телефонний довідник

```
dic = {'Іван' : "234-12-11", 'Марія' : "234-14-18"}
print("Маємо довідник:", dic)
dic['Петро'] = "405-45-34"
print("Додали абонента:",dic)
print('Роздрукували абонента "Іван":',dic['Іван'])
del dic['Марія']
print('Вилучили абонента "Марія":', dic)
print("Список абонентів без телефонів",list(dic.keys()))
```

Результат:

```
Маємо довідник: {'Іван': '234-12-11', 'Марія': '234-14-18'}
Додали абонента: {'Іван': '234-12-11', 'Марія': '234-14-
18', 'Петро': '405-45-34'}
Роздрукували абонента "Іван": 234-12-11
Вилучили абонента "Марія": {'Іван': '234-12-11', 'Петро':
'405-45-34'}
Список абонентів без телефонів ['Іван', 'Петро']
```

3.7.Стандартні функції та методи

3.7.1. Функції `any()` та `all()`

1. Функція any (<Послідовність>) повертає значення `True`, якщо в послідовності існує хоч один елемент, який у логічному контексті повертає значення `True`.

2. Якщо послідовність не містить елементів взагалі, то повертається значення False.

Приклад 3.207.

```
>>> any([0, None])
False
>>> any([])
False
>>> any([0, None, 1])
True
>>> any(["a"])
True
```

Функція **all(<Послідовність>)** повертає значення `True`, якщо всі елементи послідовності в логічному контексті повертають значення `True` або послідовність не містить елементів.

Приклад 3.208.

```
>>> all([18, "Petrenko"])
True
>>> all([])
True
>>> all([0, "Petrenko"])
False
>>> all([18, ""])
False
>>> all([0, None])
False
```

3.7.2. Перевертання й перемішування списку

Метод `reverse()`

Метод `reverse()` змінює порядок проходження елементів списку на протилежний.

Метод змінює поточний список і нічого не повертає.

Приклад 3.209.

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.reverse() # Змінюється поточний список
>>> arr
[5, 4, 3, 2, 1]
>>> arr = ["a", "b", "c", "d"]
>>> arr.reverse()
>>> arr
['d', 'c', 'b', 'a']
```

Функція-ітератор reversed

Нехай дано: arr=[1,2,3,4,5]

1. Подібність до методу `arr.reverse`

Функція `reversed(arr)`, як і метод `arr.reverse()`, змінює порядок проходження елементів списку на протилежний.

2. Відмінність від `reverse`

Дає можливість одержати новий список зі зворотним порядком: `reversed(arr)`

Формат функції:

reversed(<Послідовність>).

Властивості функції `reversed()`:

1. Функція повертає ітератор.

2. Список можна одержати за допомогою функції `list()` або генератора списків.

Приклад 3.210. Застосування функції `reversed`

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> reversed(arr)
<list_reverseiterator object at 0x02E8A530>
>>> list(reversed(arr)) # Використання list()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
# Вивід за допомогою циклу
>>> for i in reversed(arr): print(i, end=" ")
10 9 8 7 6 5 4 3 2 1
# Використання генератора списків
>>> [i for i in reversed(arr)]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

3.7.3. Сортування списків

Відсортувати список дозволяє метод **sort**().

Метод має наступний формат:

sort([key=None][, reverse= False])

Властивості методу `sort()`:

1. Усі параметри не є обов'язковими.

2. Метод змінює поточний список і нічого не повертає.

3. Параметр `key` може вказувати на функцію, що задає умови сортування

Приклад сортування за зростанням (параметр `reverse= False` за замовчуванням)

Приклад 3.211.

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort()# Змінює поточний список
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Приклад сортування за спаданням (параметр `reverse= True`)

Щоб відсортувати список за спаданням, слід в параметрі `reverse` указати значення `True`:

Приклад 3.212.

```
>>> arr = [7, 2, 10, 4, 8, 6, 9, 3, 1, 5]
>>> arr.sort(reverse = True)
>>> arr # Сортування за спаданням
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr = ["sky","land","water","fire","sun"]
>>> arr.sort(reverse = True)
>>> arr
['water', 'sun', 'sky', 'land', 'fire']
>>> arr = ["sky","land","water","fire","Sun"]
>>> arr.sort(reverse = True)
>>> arr
['water', 'sky', 'land', 'fire', 'Sun']
```

Сортування залежить від регістру

Стандартне сортування залежить від регістру символів

Приклад 3.213.

```
arr=["ant", "Asia", "bee", "Brazil"]#До сортування
```

```
arr.sort()
for i in arr:
    print(i, end=" ")
# Результат виконання: Asia Brazil ant bee
```

Щоб регістр символів не враховувався, можна вказати посилання на функцію для зміни регістру символів у параметрі `key`

Приклад 3.214.

```
arr=["ant", "Asia", "bee", "Brazil"]#До сортування
arr.sort(key=str.lower)# Указуємо метод lower()
for i in arr:
    print(i, end=" ")
# Результат виконання: ant Asia bee Brazil
```

Інші застосування параметра `key`

1. У параметрі **`key`** можна вказати функцію, що виконує будь-яку дію над кожним елементом списку.

2. Як єдиний параметр вона повинна приймати значення чергового елемента списку, а як результат – повертати результат дій над ним.

3. Цей результат буде брати участь у процесі сортування, але значення самих елементів списку не зміняться.

Приклад 3.215. Сортування по першому елементу

```
>>> def getkey(item):
        return item[0]
>>> s = [[10, 3], [1, 7], [9, 34], [3, 64]]
>>> s.sort(key=getkey)
>>> s
[[1, 7], [3, 64], [9, 34], [10, 3]]
```

4. Метод **`sort()`** сортує сам список і не повертає жодного значення.

Функція `sorted()`

Функція `sorted()` формує новий список, а поточний список залишає без змін.

sorted(<Послідовність>[, key=None] [, reverse= False])

1. Перший параметр <Послідовність> повинен містити список, який необхідно відсортувати.

2. Параметр key може вказувати на функцію, що задає умови сортування

3. Параметр [reverse= False] сортувати за зростанням.

Параметр [reverse= True] сортувати за спаданням.

Приклад 3.216. Застосування функції **sorted()**.

```
>>> arr = [7, 10, 4, 2, 6, 8, 9, 3, 1, 5]
>>>sorted(arr) # Повертає новий список!
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>sorted(arr,reverse=True) #Повертає новий список!
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
arr = ["Asia","bee", "ant", "Brazil"]
arr1 = sorted(arr,key=str.lower) # метод lower()
for i in arr1:
    print(i, end=" ")
# Результат виконання: ant Asia bee Brazil
```

Як уникнути виключення в методі join()

<Рядок>=<Роздільник>**.join**(<Послідовність>)

Уникнути виключення можна за допомогою виразу-генератора, усередині якого поточний елемент списку перетворюється в рядок за допомогою функції str():

Приклад 3.217.

```
>>> arr = ["word1", "word2", "word3"]
>>> "-".join(arr)
'word1-word2-word3'
>>> arr = ["word1", "word2", "word3", 2]
>>> "-".join((str(i) for i in arr))
'word1-word2-word3-2'
```

Крім того, за допомогою функції **str()** можна відразу одержати строкове представлення списку:

Приклад 3.218.

```
>>> arr = [ "word1", "word2", "word3", 2]
>>> str(arr)
"['word1', 'word2', 'word3', 2]"
```

3.7.4. Модуль itertools

Модуль **itertools** містить функції, що дозволяють:

- генерувати різні послідовності на основі інших послідовностей,

- виконувати фільтрацію елементів і ін.

Усі функції повертають об'єкти, що підтримують ітерації (ітератори).

Перш ніж використовувати функції, необхідно підключити модуль за допомогою інструкції:

```
import itertools
```

Генерація невизначеної кількості значень

Для генерації невизначеної кількості значень призначені наступні функції:

Функція **count([start=0][, step=1]).**

Створює нескінченно наростаючу послідовність значень. Початкове значення задають параметром `start`, а крок –параметром `step`.

Приклад 3.219.

```
import itertools
for i in itertools.count():
    print(i)
    if i % 10 == 0:
        check= input("Продовжуєм?")
        if check != "": break
```

Виводимо по 10 елементів. Використання функції `count` з функцією `zip`.

```
import itertools
```

```
m=input("Введіть послідовність символів")
p = list(zip(itertools.count(),m ))
print(p)
```

Результат:

```
[(0,'а'), (1,'б'), (2,'в'), (3,'г'), (4,'д')]
```

```
import itertools
m=input("Введіть послідовність символів")
p = list(zip(itertools.count(start=2, step=2),m ))
print(p)
```

Результат:

[(2,'а'), (4,'б'), (6,'в'), (8,'г'), (10,'д')]

Функція **cycle(<Послідовність>)**

На кожній ітерації повертається черговий елемент послідовності.

Коли буде досягнутий кінець послідовності, перебір почнеться спочатку, і так нескінченно.

Приклад 3.220.

```
import itertools
n = 1
for i in itertools.cycle("абв"):
    if n > 10: break
    print(i, end=" ")
    n += 1
```

Результат:

```
а б в а б в а б в а
```

```
import itertools
p = list(zip(itertools.cycle([0, 1]), "абвгд"))
print(p)
```

Результат:

[(0, 'а'), (1, 'б'), (0, 'в'), (1, 'г'), (0, 'д')]

```
import itertools
p = list(zip(itertools.cycle(["a", "b", "v", "g", "d"]),
"абвгд"))
print(p)
```

Результат: [('a', 'а'), ('b', 'б'), ('v', 'в'), ('g', 'г'), ('d', 'д')]

Функція **repeat**(<Об'єкт>[, <Кількість повторів>])

Повертає об'єкт зазначену кількість раз. Якщо кількість повторів не зазначена, то об'єкт повертається нескінченно.

Приклад 3.221.

```
import itertools
p = list(itertools.repeat(1,10))
print(p)
```

Результат:

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
p = list(zip(itertools.repeat(5), "абвгд"))
print(p)
```

Результат: [(5, 'а'), (5, 'б'), (5, 'в'), (5, 'г'), (5, 'д')]

Ітератори, що підтримують скінченні послідовності

https://docs.python.org/3/library/itertools.html

```
from itertools import*
```

Функція **chain(p,q,l,…)# *Конкатенуємо послідовності***

```
p=[1,2,3,]
q=["a","b","c","d"]
l=[7,8]
print(list(chain(p,q,l)))
```

Результат: [1, 2, 3, 'a', 'b', 'c', 'd', 7, 8]

Функція **compress(data,selector)**

```
data="ABCDE"
selector=[1,0,True,0,23]
print(list(compress(data,selector)))
# Якщо елемент списку selector повертає True
```

Результат: ['A', 'C', 'E']

Функція **islice(**seq, [start,] stop [, step]**)**

```
sec="ABCDE"
print(list(islice(sec,0,4,2)))
# Вибирає як range з послідовності sec
['A', 'C']
```

Функція **product(seq, repeat=2)** (Комбінаторні функції)

```
from itertools import*
print(list(product('AB', repeat=2)))
[('A','A'), ('A','B'), ('B','A'), ('B','B')]
from itertools import*
print(list(product('01', repeat=3)))
[('0','0','0'), ('0','0','1'), ('0','1','0'), ('0','1','1'), ('1','0','0'), ('1','0','1'), ('1','1','0'), ('1','1','1')]
```

3.8. Функції користувача

Функція – це фрагмент коду, який можна викликати з будь-якого місця програми **більше одного разу**.

Ми вже багато разів використовували, так звані, вбудовані функції мови Python.

Наприклад:

Функція `len()` – дозволяє одержати кількість елементів послідовності.

Функція `str()` – перетворює будь-який об'єкт у рядок.

Розглянемо створення функцій користувача

Функції дозволяють:

зменшити надмірність програмного коду та

підвищити його структурованість.

3.8.1. Визначення функції і її виклик

Функцію створюють (або, як говорять програмісти, визначають) за допомогою ключового слова

def за наступною схемою:

Приклад 3.222.

```
def <Ім'я функції> ([<Параметри>]):
    [""" Рядок документування """]
    <Тіло функції>
    [return <Результат>]
```

<Ім'я функції> повинно бути унікальним ідентифікатором, який формується за такими правилами:

1. Ім'я функції може містити:

латинські букви, цифри, знаки підкреслення.

2. Ім'я функції не може починатися

з цифри, не можна використовувати ключові слова, слід уникати збігів з назвами вбудованих ідентифікаторів.

3. Регістр символів у назві має значення.

Після імені записують параметри([<Параметри>]) функції в круглих дужках.

1. Можна вказати один або кілька параметрів через кому (q,z)

2. Якщо функція не приймає параметри, вказують тільки круглі дужки ()

3. Після круглих дужок ставлять двокрапку.

```
def <Ім'я функції> ([<Параметри>]):
```

Наприклад: `def func(a,b):`

Тіло функції є складеною конструкцією.

1. Інструкції всередині функції

виділяють однаковою кількістю пробілів ліворуч.

2. Кінцем функції вважається інструкція,

перед якою перебуває менша кількість пробілів.

3. Якщо тіло функції не містить інструкцій, то усередині неї необхідно розмістити оператор `pass`, який не виконує ніяких дій.

Оператор **`pass`** – оператор зручно використовувати на етапі налагодження програми, коли функція визначена, а тіло вирішено дописати пізніше. Приклад функції, яка нічого не робить:

Приклад 3.223.

```
def func():
    pass
```

Необов'язкова інструкція `return` дозволяє повернути з функції яке-небудь значення як результат.

```
def func(a):
```

```
        return a
    def func():
        a=10
        return a
```

1. Після виконання цієї інструкції виконання функції буде зупинено.

2. Це означає, що інструкції, що слідують після оператора **return**, ніколи не будуть виконані.

Приклад 3.224.

```
def func():
    print("Текст до інструкції return")
    return "значення, що повертається"
    print("Ця інструкція ніколи не буде виконана")
print(func()) # Викликаємо функцію
```

Результат виконання:

Текст до інструкції return

значення, що повертається

Інструкції **return** може не бути. У цьому випадку виконуються всі інструкції усередині функції, і як результат повертається значення **None**.

Приклад 3.225.

```
def print_ok():
    """ Приклад функції без параметрів """
    print("Повідомлення при вдало виконаній операції")
def echo(m) :
    """ Приклад функції з параметром """
    print(m)
def suma(x, y) :
    """Приклад функції з параметрами,
       що повертає суму двох змінних"""
    return x + y
```

Виклик функції

Виклик функції записують у форматі

```
<Ім'я функції> ([<Параметри>])
```

1. При виклику функції значення передають через параметри усередині круглих дужок.

2. Параметри записуються через кому.

3. Якщо функція не приймає параметрів, то вказують тільки круглі дужки.

4. Кількість параметрів у визначенні функції повинна збігатися з кількістю параметрів при виклику, інакше буде виведене повідомлення про помилку.

Приклад 3.226.

```
def prok():
    """ Приклад функції без параметрів """
    print("Повідомлення при вдало виконаній операції")
prok()
```

Результати роботи:

Повідомлення при вдало виконаній операції

```
def sumfunc(x, y) :
    """Приклад функції з параметрами,
       що повертає суму двох змінних"""
    return x + y
x = sumfunc(5, 2)
print(x)
```

Результати роботи: 7

```
m="довільний рядок"
def echo(m) :
    """ Приклад функції з параметром """
    print(m)
print(echo(m))
```

Результати роботи:

довільний рядок

```
None
```

3.8.2. Області видимості змінних

1. Ім'я змінної у виклику функції **може не збігатися** з іменем змінної у визначенні функції.

Приклад 3.227.

```
def echo(m) :
    print(m)
n="Параметр має інше ім’я"
echo(n)
```

Результати роботи:

Параметр має інше ім’я

2. Крім того, *глобальні* змінні **x** и **y** не конфліктують з однойменними змінними у визначенні функції, оскільки вони розташовані в різних областях видимості.

Приклад 3.228.

```
def sumfunc(x, y) :
    return x + y
x = 20
y = 45
z = x + y
print(z)
a = 10
b = 11
z= sumfunc(a,b)
print(z)
```

Результат:

65

21

Функція **може приймати ті типи значень**, які відповідають операторам в її тілі.

Оператор +, використовуваний у функції `suma()`, застосовується не тільки для додавання чисел, але й дозволяє об'єднати послідовності.

Тобто, функція `suma()` може використовуватися не тільки для додавання чисел.

Приклад 3.229.

```
def suma(x, y):
    return x + y
print(suma("str", "ing"))
print(suma([1, 2], [3, 4]))
print(suma(2,3))
```

Результат роботи:

string

[1, 2, 3, 4]

5

3.8.3. Функція як об'єкт мови Python

У мові `Python` усі елементи є об’єктами: рядки, списки, кортежі, типи даних і функції.

Інструкція **`def`** створює об'єкт, що має тип **`function`**, і зберігає посилання на нього в ідентифікаторі, зазначеному після інструкції **`def`**.

Таким чином, ми можемо зберегти посилання на функцію в інший змінній – для цього назву функції вказують без круглих дужок.

Ми можемо зберігати це посилання в змінній і викликати функцію через неї.

Приклад 3.230. Збереження посилання на функцію в змінній

```
def suma(x, y):
    return x + y
f = suma      # Зберігаємо посилання в змінній f
v = f(10, 20) # Викликаємо функцію через змінну f
print("10 + 20 =",v)
```

Результат роботи: 10 + 20 = 30

1. У цьому прикладі об'єкт типу `function` збережений у змінній **`f`** за допомогою інструкції:

```
f = suma
```

2. Змінну **`f`** можемо використовувати для виклику функції `suma`:

```
v = f(10, 20)
```

Посилання на функцію може бути передане як параметр в іншу функцію

Функції, передані за посиланням, зазвичай називають *функціями зворотного виклику.*

Приклад 3.231. Функції зворотного виклику

```
def s (z, y):
  """Функція s()"""
      return z + y
def func(fparam):
    a = 10
    b = 20
    return fparam(a, b) # Викликаємо функцію s()
# Передаємо посилання на функцію як параметр
v = func(s)
print("suma=", v)
```

Результат роботи: `suma= 30`

3.8.4. Атрибути об'єкта типу `function`

Звернутися до атрибутів можна, указавши атрибут після назви функції через крапку.

Приклад 3.232.

Через атрибут `_name_` можна одержати назву функції у вигляді рядка.

```
print(s.__name__)
print(func.__name__)
s
func
```

Через атрибут _doc_ – рядок документування і т. д.

```
print(suma.__doc__)
print(func.__doc __)
```

Приклад 3.233. Атрибути функції suma

```
>>> def suma(x, y):
    """ Додавання двох чисел """
    return x + y
```

Приклади виклику атрибутів функції

```
# Одержуємо ім'я функції
>>> suma.__name__
suma
# Одержуємо рядок документування
>>> suma.__doc__
Додавання двох чисел
# Одержуємо параметри функції
print(suma.__code__.co_varnames)
('x', 'y')
# Довідаємося, в якому модулі розташована функція
print(suma.__module__)
__main__
```

1. Усі інструкції в програмі виконуються послідовно зверху вниз.

2. Тому, перш ніж використовувати в програмі ідентифікатор, його необхідно попередньо визначити, присвоївши йому значення.

3. Отже, визначення функції повинно бути розташоване перед викликом функції.

Приклад 3.234.

Правильно:

```
def difference (x, y) :
    return x - y
v = difference(20, 9)# Викликаємо після визначення.
```

```
print("20 - 9 =",v)
```

Результат роботи: 20 - 9 = 11

Неправильно:

```
v = difference(20, 9)  # Ідентифікатор ще не визначений. Це помилка!!!
def difference(x, y):
    return x - y
```

Результат роботи:

```
Traceback (most recent call last):
  File "C:/PYTHON/Samples.py", line 1, in <module>
    v = difference(20, 9)
NameError: name 'difference' is not defined
```

Щоб уникнути помилки, визначення функції розміщають на початку програми після підключення модулів або в окремому модулі.

3.8.5. Кілька функцій з однією назвою

За допомогою оператора розгалуження **if** можна вибирати необхідне визначення функції з однаковою назвою, але різною реалізацією.

Приклад 3.235.

```
n = input("Введіть 1 для виклику першої функції: ")
if n == "1" :
    def echo():
        print("Ви ввели число 1")
else:
    def echo():
        print("Альтернативна функція")
echo() # Викликаємо функцію
```

Результат роботи1:

Введіть 1 для виклику першої функції: 1

Ви ввели число 1

Результат роботи2:

Введіть 1 для виклику першої функції: 0

Альтернативна функція

Випадок перевизначення функції

Пом’ятайте, що інструкція **def** усього лише присвоює посилання на об'єкт функції ідентифікатору, розташованому після ключового слова **def**.

Якщо визначення однієї функції зустрічається в програмі кілька разів, то буде використовуватися функція, яка була визначена останньою.

Приклад 3.236.

```
def echo():
    print("Ви ввели число 1")
def echo():
    print("Альтернативна функція")
echo() # Завжди виводить "Альтернативна функція"
```

Результат роботи: Альтернативна функція.

3.8.6. Необов'язкові параметри й зіставлення по ключах

1. Щоб зробити деякі параметри необов'язковими, слід у визначенні функції присвоїти цьому параметру початкове значення.

2. Необов'язкові параметри повинні слідувати після обов'язкових параметрів, інакше буде виведене повідомлення про помилку.

Приклад 3.237. Необов'язкові параметри

```
def suma (x, y, z=2): #z-необов'язковий параметр
    return  x + y + z
a = suma(5,10)
print("a =",a)
b = suma (10, 50, 40)
print("b =",b)
```

Результат роботи: a = 17

b = 100

Таким чином, якщо третій параметр не заданий, то його значення буде дорівнювати 2.

Позиційна передача параметрів – це присвоєння значень параметрам функції в порядку, у якому вони задані при виклику функції.

Приклад 3.238. Позиційна передача параметрів

```
def forcalc(x,y,z):
    print("x =",x, "y =",y, "z =",z)
    return x*y+z
print("m =", forcalc(3,5,7))
```

Результат роботи: x = 3 y = 5 z = 7

m = 22

Змінній **x** при зіставленні буде присвоєно значення 3, змінній **y** – значення 5, а змінній **z** = 7.

3.8.7. Передача параметрів зіставленням по ключах

1. При виклику функції параметрам можуть присвоюватися значення.

2. Послідовність вказівки параметрів у цьому випадку може бути довільною.

Приклад 3.239. Зіставлення по ключах

```
def forcalc(x,y,z):
    print("x =",x, "y =",y, "z =",z)
    return x*y+z
print("m =", forcalc(z=7, x=3, y=5))
```

Результат роботи: x = 3 y = 5 z = 7

m = 22

Зіставлення по ключах і необов'язкові параметри

Зіставлення по ключах дуже зручно використовувати, якщо функція має кілька необов'язкових параметрів.

У цьому випадку не потрібно перераховувати всі значення, а достатньо присвоїти значення потрібному параметру.

Приклад 3.240.

```
def suma(a=2, b=3, c=4): # Усі параметри необов'язкові
    return a + b + c
print("case1",suma())          #За замовчуванням
```

```
    print("case2",suma(2, 3, 20))#Позиційне присвоювання
print("case3",suma(c = 15))  #Зіставлення по ключах
```

Результат роботи: case1 9

case2 25

case3 20

3.8.8. Розпакування списку або кортежу

Якщо значення параметрів, які планується передати у функцію, містяться в кортежі або списку, то перед об'єктом слід указати символ * (*розпакувати*).

Розглянемо передачу значень із кортежу й списку.

Приклад 3.241.

```
def suma (a, b, c):
    return a + b + c
t1 = (1, 2, 3)
arr =[6, 7, 8]
# Розпакували кортеж
print("case1",suma(*t1))
# Розпакували список
print("case2",suma(*arr))
```

Результат роботи: case1 6

case2 21

Суміщення параметрів з розпаковкою

```
def suma (a, b, c):
    return a + b + c
t2 = (4, 5) # Розпакування на два параметри з 3
d=1
print("case3",suma(d, *t2)) # Комбінувати
Результат роботи: case3 10
t1 = (1, 2, 3)
arr =[6, 7, 8]
print("case4",suma(t1,t2,tuple(arr)))#without
```

```
Результат роботи: case4 (1, 2, 3, 4, 5, 6, 7, 8)
```

Розпакування словника

Якщо значення параметрів містяться в словнику, то розпакувати словник можна, указавши перед ним дві зірочки: **(* *)**.

Приклад 3.242.

```
def suma(a, b, c):
    return a + b + c
d1 = {"a": 1, "b": 2, "c": 3}
print("dict", suma(**d1)) # Розпаковуємо словник
```

Результат роботи: dict 6

```
t  = (5, 6)
d2 = {"c": 3}
print("comb",suma(*t, **d2))#Можна комбінувати значення
```

Результат роботи: comb 14

Передача у функцію незмінюваних об'єктів. Об'єкти у функцію передають за посиланням.

Якщо об'єкт є об’єктом незмінюваного типу, то зміна значення усередині функції не торкнеться значення змінної поза функцією:

Приклад 3.243.

```
def func(a, b):
    a, b = 20, "str"
    print("a =",a, "b =",b)
x, s = 80, "test"
func(x, s)
print("x =",x, "s =",s)
```

Результат роботи: a = 20 b = str

x = 80 s = test

У цьому прикладі значення в змінних x і s не змінилися. Однак якщо об'єкт є об’єктом змінюваного типу, то ситуація буде іншою:

Передача у функцію змінюваних об'єктів

Приклад 3.244.

```
def func (a):
    a[0] = "str"
x = [1, 2, 3]
    func (x)
print("x =",x )
```

Результат: x = ['str', 2, 3]

```
def func (b):
    b["a"] = "str", 800
y = {"a": 1, "b": 2}
func (y)
print("y =",y)
```

Результат:`y = {'a': ('str', 800), 'b': 2}`

Як видно з прикладу, значення в змінних **x** та **y** змінилися, оскільки список і словник є об’єктами змінюваних типів.

Якщо необхідно уникнути зміни значень, усередині функції слід створити копію об'єкта.

Приклад 3.245. Передача змінюваного об'єкта у функцію

```
def func(a, b):
    a = a[:] # Створюємо поверхневу копію
    b = b.copy() # Створюємо поверхневу копію
    a[0], b["a"] = "str", 800
    print("a =",a, "b =",b)
x = [ 1, 2, 3] # Список
y= {"a": 1, "b": 2} # Словник
func(x, y) # Значення залишаться колишніми
print("x =",x, "y =",y)
```

Результат роботи:

a = ['str', 2, 3] b = {'a': 800, 'b': 2} x = [1, 2, 3] y = {'a': 1, 'b': 2}

Можна також відразу передавати копію об'єкта у виклику функції:

Приклад 3.246.

```
def func(a, c):
    print(a)
    print(c)
    a[0], c["a"] = "str", 800
    print("a =",a, "b =",c)

x = [ 1, 2, 3] # Список
y = {"a": 1, "b": 2} # Словник
func(x[:], y.copy()) # Значення залишаться колишніми
print("x =",x, "y =",y)
```

Результат роботи:

a = ['str', 2, 3] c = {'a': 800, 'b': 2}

x = [1, 2, 3] y = {'a': 1, 'b': 2}

3.8.9. Значення змінюваного типу за замовчуванням

Якщо вказати об'єкт, що має змінюваний тип, як значення за замовчуванням, то цей об'єкт буде зберігатися між викликами функції.

Приклад 3.247.

```
def func(a = []):
    a.append(2)
    return a
print(func()) # Виведе: [2]
print(func()) # Виведе: [2, 2]
print(func()) # Виведе: [2, 2, 2]
```

Результат роботи: [2]

[2, 2]

[2, 2, 2]

Як видно з прикладу, значення накопичуються усередині списку.

Як уникнути накопичення

Уникнути накопичення можна у такий спосіб:

Приклад 3.248.

```
def func (a=None):
    # Створюємо новий список, якщо значення дорівнює None
    if a is None:
       a = []
    a.append(2)
    return a
print(func())
print(func([1]))
print(func([1,3]))
print(func()) # Виведе: [2]
```

Результат роботи: [2]

[1, 2]

[1, 3, 2]

[2]

3.8.10. Складність функцій

Функція може бути будь-якої складності і повертати будь-які об'єкти (списки, кортежі, і навіть функції!):

Приклад 3.249.

```
def outfunc(n):
   def infunc(x):
       return x + n
    return infunc
new = outfunc(100)  # new - это функция
print(new(200))
```

Результат:

300

Приклад 3.250. Прикладна функція ділення

```
def safe_div(x, y):
    """Do a safe division
       for fun and profit"""
    if y != 0:
        z = x / y
        print (z)
    else:
        print ("Помилка ділення на нуль")
         exit(0)
safe_div(1,2)
safe_div(1,0)
safe_div(2,2)
```

Результат: 0.5

Помилка ділення на нуль

Обчислення числа Фібоначчі

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, …

Послідовність чисел Фібоначчі $\{F_n\}_{n=0}^{\infty}$ **задамо рекурентно:**

$$F_0, F_1, F_n = F_{n-1} + F_{n-2}, n \geq 2, n \in Z$$

Приклад 3.251.

```
def fibb(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fibb(n-1) + fibb(n-2)
print(fibb(2))
```

Результат: 1

```
print(fibb(3))
```

Результат:2

```
print(fibb(4))
```

Результат: 3

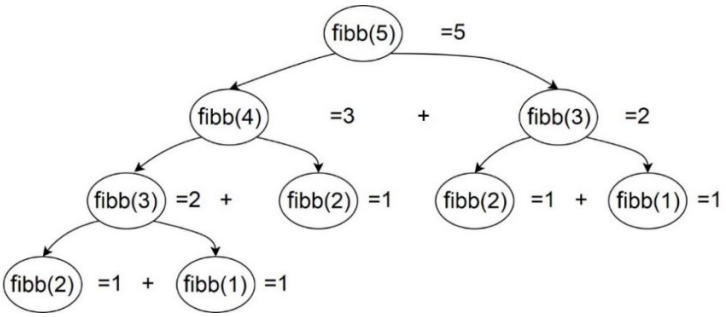


Рис. 3.2. Граф виконання рекурсії

Приклад 3.252.

Обчислення факторіала

```
def factorial(n):
    prod = 1
    for i in range(1, n + 1):
        prod *= i
    return prod
print(factorial(10))
```

Результат: 3 628 800

```
print(factorial(50))
```

Результат:

30414093201713378043612608166064768844377641568960512000000000000

3.8.11. Фіксоване число параметрів у функції

Число параметрів, які використовуються у функції, дорівнює числу параметрів, які вказані у круглих дужках.

Приклад 3.253.

```
def fixparam(a,b,c):
    return a+b-c
print(fixparam(1,2,3))
```

Результат: 0

Приклад 3.254.

```
def fixparam(a,b,c=3):
    return a+b-c #a=1,b=2,c=3
m=[1,2]
d=fixparam(*m)
print(d)
```

Результат: 0

Змінне число параметрів у функції

1. Якщо перед параметром у визначенні функції вказати символ (*), то функції можна буде передати довільну кількість параметрів.

2. Усі передані параметри будуть об’єднані в кортеж. Для прикладу напишемо функцію додавання довільної кількості чисел.

Приклад 3.255. Збереження переданих даних у кортежі

```
def suma(*t):
    """ Приймаємо довільну кількість параметрів """
    print("t=",t)
    res=sum(t)
    return res
print("Сума=",suma(10, 20, 30, 40, 50))
```

Результат роботи: t= (10, 20, 30, 40, 50)

Сума= 150

Приклад 3.256. Збереження кортежів

```
def f(*t): print(t)
    d=(1,2,3)
    f(*d)
```

Результат роботи: (1, 2, 3)

3.8.12. Поєднання обов’язкових параметрів з зірковим

Спочатку вказують обов'язкові (позиційні) параметри, а останнім вказують параметр з зірочкою:

Приклад 3.257.

```
def fixmix(a,b,*t):
    return a + b + t
m=5
k=(1,2)
n=(3,4)
l=6
d=fixmix(k, n, m, l)
print(d)
```

Результат: (1, 2, 3, 4, 5, 6)

Обов'язкові параметри зі значеннями за замовчуванням

Спочатку вказують обов'язкові параметри, потім необов’язкові параметри, а останнім вказують параметр з зірочкою:

Приклад 3.258.

```
def sumator(x, y=5, *t): # Комбінація параметрів
    res = x + y
    for i in t:
        res += i
    return res
print(sumator(10))
# Перебираємо кортеж з переданими параметрами
print(sumator(10, 20, 30, 40, 50))
```

Результат роботи: 15

150

3.8.13. Збереження іменованих параметрів у словнику

Якщо перед параметром у визначенні функції вказати дві зірочки: (**), то всі іменовані параметри будуть збережені в словнику.

Приклад 3.259. Збереження переданих даних у словнику

```
def func(**d):
    print(d)  #запакувати
func(a=1, b=2, c=3)
Результат: {'c': 3, 'a': 1, 'b': 2}
```

```
def func(**d):
    print(d)    #запакувати
    m={"a":1,"b":2}
    func(**m)       #розпакувати
```

Результат:`{'b':2, 'a':1}`

Поєднання обов'язкових параметрів з двозірковим

Спочатку вказують обов'язкові параметри, а останнім вказують двозірковий параметр:

Приклад 3.260.

```
def dicmix(a,**d):
    print(a, d)
m=5
d=dicmix(m,k=1,b=2,c=3)
```

Результат: 5 {'k': 1, 'b': 2, 'c': 3}

Приклад 3.261.

```
def dicnmix(a,b=2,**d):
    print(a,b,d)
m=5
d=dicnmix(m,k=1,g=2,c=3)
```

Результат: 5 2 {'k': 1, 'g': 2, 'c': 3}

Комбінування (*) і (**)

При комбінуванні параметрів параметр з двома зірочками вказується останнім.

Якщо у визначенні функції вказується комбінація параметрів з однією зірочкою й двома зірочками, то функція прийме будь-які передані їй параметри.

Приклад 3.262. Комбінування параметрів

```
def func(*t, **d):
    """ Функція прийме будь-які параметри """
    print(t)
    print(d)
func(35, 10, a=1, b=2, c=3)
```

Результат роботи:

(35, 10)

{'a': 1, 'b': 2, 'c': 3}

```
def func(*t, **d):
    """ Функція прийме будь-які параметри """
    print(t)
    print(d)
func(10)
```

Результат роботи:

(10,)

{}

```
func(a= 1, b=2)
```

Результат роботи:

()

{'a': 1, 'b': 2}

Поєднання обов'язкових параметрів з зірковими

Спочатку вказують обов'язкові параметри, потім однозірковий параметр, а останнім вказують двозірковий параметр.

Приклад 3.263.

```
def func(g, *t, **d):
    """ Функція прийме будь-які параметри """
    print(g)
    print(t)
    print(d)
func(35, 10, 11, a=1, b=2, c=3)
```

Результат:

35

(10, 11)

{'a': 1, 'b': 2, 'c': 3}

Передача параметрів тільки по іменах

За умови, коли параметри передаються тільки по іменах, їх розміщують між однозірковим параметром та двозірковим параметром.

Тобто параметри повинні вказуватися після параметра з однією зірочкою, але перед параметром з двома зірочками.

Іменовані параметри можуть бути необов'язковими

Такі параметри мають значення за замовчуванням.

Приклад виклику функції:

```
func(*t, a, b=10, **d)
```

де `*t` – однозірковий параметр,

`a` – обов'язковий параметр, який обов'язково задають по імені при виклику функції,

`b` – необов'язковий параметр, який можна не задавати або задавати тільки по імені при виклику функції,

`**d` – двозірковий параметр.

Приклад 3.264. `a, b` – тільки по іменах

```
def func(*t, a, b=10, **d):
    print(t, a, b, d)

func(35, 10, a=1, c=3, k=22)
```

Результат роботи:(35, 10) 1 10 {'c': 3, 'k': 22}

```
func (11, 12, a=5, b=4)
```

Результат роботи: (11, 12) 5 4 {}

```
func(a=1, b=2)
```

Результат роботи: () 1 2 {}

```
func (1, 2, 3)
```

TypeError: func() missing 1 required keyword-only argument: 'a'

Пояснення правил виклику функції у випадку, коли при визначенні функції обов'язкові або необов'язкові параметри вказані між однозірковим та двозірковим параметрами

1. У цьому випадку змінна **t** прийме будь-яку кількість значень, які будуть об'єднані в кортеж.

Наприклад: 35,10

2. Змінні **a** й **b** повинні передаватися тільки по іменах, причому змінній **a** обов'язково потрібно передати значення при виклику функції.

Наприклад: `a=1`

3. Змінна **b** має значення за замовчуванням, тому при виклику допускається не передавати їй значення, але якщо значення передається, то воно повинно також передаватися по імені.

Наприклад: `b=1`

4. Змінна **d** прийме будь-яку кількість іменованих параметрів і збереже їх у словнику.

Наприклад: `c=3, k=22`

5. Хоча змінні **a** й **b** є іменованими, вони не потраплять у цей словник.

Наприклад: `{'c': 3, 'k': 22}`

6. Параметра із двома зірочками може не бути у визначенні функції

Наприклад: 11, 12, a=5, b=4

7. Параметр з однією зірочкою при вказівці параметрів, переданих тільки по іменах, повинен бути присутнім обов'язково.

Випадок, коли функція не повинна приймати змінну кількість параметрів, але необхідно використовувати змінні, передані тільки по іменах

У цьому випадку вказують тільки зірочку без змінної:

Приклад 3.265.

```
def func ( x=1, y=2, *, a, b=10):
    print(x, y, a, b)
    func (35, 10, a=1)
```

Результат роботи: 35 10 1 10

```
func (10, a=5)
```

Результат роботи: 10 2 5 10

```
func (a=1, b=2)
```

Результат роботи: 1 2 1 2

```
func (a=1, y=8, x=7)
```

Результат роботи:7 8 1 10

```
func (1, 2, 3)
```

TypeError:

Пояснення правил виклику функції

1. У цьому прикладі значення змінним **x** и **y** можна передавати як по позиціях, так і по іменах.

Наприклад: 35,10 або `y=8, x=7`

2. Оскільки змінні мають значення за замовчуванням, допускається взагалі не передавати їм значень.

Наприклад: `func (a=1, b=2)`

3. Змінні **a** й **b** розташовані після параметра з однією зірочкою, тому передати значення при виклику можна тільки по іменах.

Наприклад: `func (a=1, b=2)`

4. Оскільки змінна **b** має значення за замовчуванням, допускається не передавати їй значення при виклику, а змінна **a** обов'язково повинна одержати значення, причому тільки по імені.

Наприклад: `func(10, a=5)`

3.8.14. Анонімні функції

Крім звичайних, мова Python дозволяє використовувати анонімні функції, які також називають лямбда-функціями. Анонімну функцію описують за допомогою ключового слова **lambda** за наступною схемою:

lambda [<Параметр1>[,...,<Параметрn>]]:<значення, що повертається>

Після ключового слова **lambda** можна вказати <параметри>.

У полі <значення, що повертається> вказують вираз, результат виконання якого буде повернутий функцією.

Анонімна функція не має імені

Визначення анонімної функції повертає посилання на об'єкт-функцію, яку можна зберегти в змінній або передати як параметр в іншу функцію.

Приклад 3.266.

```
def func(b):
    return b(2)
print(func(lambda x: x**2))
```

Результат: 4

Викликати анонімну функцію можна, як і звичайну, за допомогою круглих дужок, усередині яких розташовані передані параметри.

```
a=lambda x: x**2
print(a(2))
```

Розглянемо приклад використання анонімних функцій.

Приклад 3.267. Приклад використання анонімних функцій

Визначення lambda-функцій

```
# Функція без параметрів
f1 = lambda: 10 + 20
# Функція з двома параметрами
f2 = lambda x, y: x + y
# Функція з трьома параметрами
f3 = lambda x, y, z: x + y + z
```

Виклик lambda-функцій

```
print(f1())
Результат роботи: 30
print(f2(5, 10))
Результат роботи: 15
print(f3(5, 10, 30))
```

Результат роботи: 45

Обов'язкові та необов'язкові параметри

Як і у звичайних функціях, деякі параметри анонімних функцій можуть бути необов'язковими. Для цього параметрам у визначенні функції присвоюється значення за замовчуванням.

Приклад 3.268. Необов'язкові параметри в анонімних функціях

Визначення lambda-функції

```
f = lambda x, y = 2: x + y
```

Виклик lambda-функції

```
print (f(5))
```

Результат роботи: 7

```
print(f(5, 6))
```

Результат роботи: 11

Передача анонімної функції як параметра

Найчастіше анонімну функцію не зберігають у змінній, а відразу передають як параметр в іншу функцію.

Наприклад, метод списків `sort()` дозволяє вказати функцію користувача в параметрі `key`.

Приклад 3.269. Сортування по четвертому символу

```
arr=["anaconda", "Africa", "beer", "Brazil"]
arr.sort(key=lambda s: s[3])
for i in arr:
    print(i, end=" ")
```

Результат роботи: anaconda Africa beer Brazil

Приклад 3.270. Застосування `lambda`-функції

```
bigrams = {"AB": [10, 11, 12],
           "BC": [5, -5, 8],
           "CD": [105, 1, 0],
           "DE": [6, 6],
           "EF": [15, 20, 15],
           "FG": [22, 11, 32],
           "GH": [20, 20, 20]}
```

```
sorter = sorted(bigrams, key=lambda mk: sum(bigrams[mk]),
reverse=True)
for a in sorter:
    print(a,"-", bigrams[a])
```

Результат:

```
CD - [105, 1, 0] # Найбільша сума
FG - [22, 11, 32]
GH - [20, 20, 20]
EF - [15, 20, 15]
AB - [10, 11, 12]
DE - [6, 6]
BC - [5, -5, 8] # Найменша сума
```

Другий спосіб сортування

```
from functools import partial
def sort_func(c, dict):
    return sum(dict[c])
bigrams = {"AB": [10, 11, 12],
           "BC": [5, -5, 8],
           "CD": [105, 1, 0],
           "DE": [6, 6],
           "EF": [15, 20, 15],
           "FG": [22, 11, 32],
           "GH": [20, 20, 20]}
partial_sort = partial(sort_func, dict=bigrams)
sorter = sorted(bigrams.keys(), key=partial_sort,
reverse=True)
for a in sorter:
    print(a, bigrams[a])
```

3.8.15. Функції-генератори

Функцією-генератором називають функцію, яка може повертати лише одне значення з множини допустимих значень на кожній ітерації.

Основна ознака функції-генератора – ключове слово **yield**

```
def func(a):
    yield a
```

Правила застосування

1. До функції-генератора не можна звертатися, як до звичайної функції.

2. Функція-генератор може використовуватися лише як ітератор.

Приклад 3.271. Піднесення елементів послідовності до степеня

```
def func(x, y):
    for i in range(1, x+1):
        yield i ** y
for n in func(10, 2):
    print(n, end=" ")
```

Результат роботи:

1 4 9 16 25 36 49 64 81 100

```
for n in func(10, 3):
    print(n, end=" ")
```

Результат роботи:

1 8 27 64 125 216 343 512 729 1000

Метод `_next_()` та функції-генератори

Метод **`_next_()`** – це метод об'єктів-ітератоів

Коли значення закінчуються, метод викликає виключення `StopIteration`.

Виклик методу `_next_()` у циклі `for` проводиться непомітно для нас.

Приклад 3.272. Використання методу `_next_()`

```
def func (x, y) :
    for i in range(1, x+1):
        yield i ** y
i = func(3, 3)
print(i.__next__())
print(i.__next__())
print(i.__next__())
```

```
print(i.__next__())
```

Результат роботи:

1

8

27

```
Stopiteration
```

Відмінності звичайної функції та функції-генератора

1. Звичайна функція може повернути всі значення відразу у вигляді списку.

2. Функція-генератор повертає тільки одне значення за раз, як ітератор.

Перевага. Ця особливість дуже корисна при обробці великої кількості значень, тому що не потрібно завантажувати увесь список зі значеннями.

Функція – генератор виконує обробку даних «на льоту».

Виклик функції-генератора з функції генератора

Для цього використовуємо розширений синтаксис ключового слова **yield**:

```
yield from <Викликувана функція-генератор>
```

Розглянемо наступний приклад.

Нехай у нас є список чисел `b`

Необхідно одержати інший список, що включає числа в діапазоні від 1 до кожного із чисел у першому списку.

```
b=[2,3,4]
d=[1,2,1,2,3,1,2,3,4]
```

Щоб створити такий список, розглянемо код на прикладі.

Приклад 3.273. Виклик однієї функції-генератора з іншої (простий випадок)

```
def gen(b):
    for e in b:
        yield from range(1, e + 1)
b = [5, 10]
for i in gen(b): print(i, end = " ")
```

Результат роботи:

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

```
b = [2, 3, 4]
for i in gen(b): print(i, end = " ")
```

Результат роботи:

1 2 1 2 3 1 2 3 4

Помножимо числа списку на 2. Код, що виконує цю задачу, показаний на прикладі.

Приклад 3.274. Виклик однієї функції-генератора з іншої (складний випадок)

```
def gen2(n):
    for e in range(1, n + 1):
        yield e * 2
def gen(m):
    for i in m:
        yield from gen2(i)
n = [5, 10]
for i in gen(n): print(i, end = " ")
```

Результат роботи:

2 4 6 8 10 2 4 6 8 10 12 14 16 18 20

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

3.8.16. Декоратори функцій

Декоратори дозволяють змінити поведінку звичайних функцій – наприклад, виконати які-небудь дії перед виконанням та після виконання функції.

Приклад 3.275.

```
def my_decorator(function_to_decorate):
 def wrapper_function():
    print("Тут пишемо код, до виклику функції")
    function_to_decorate() # Сама функція
    print("Тут пишемо код, що працює після виклику")
        # Повертаємо цю функцію
 return wrapper_function
@my_decorator
```

```
def unchangeable_function():
    print("Це базова функція")
unchangeable_function()
```

Приклад 3.275. Декоратори функцій

```
def deco(f):  # Функція-декоратор
# Пишемо який завгодно код
    print("Тут можна писати код декоратора")
    return f  # Повертаємо посилання на функцію
@deco
def func(x):
    print("А тепер запустили функцію func()")
    return "x = {0}".format(x)
print(func(10))
```

Результат роботи:

Тут можна писати код декоратора

А тепер запустили функцію func()

x = 10

У цьому прикладі перед визначенням функції `func()` вказується назва функції **`deco()`** із символом @: **`@deco`**

Таким чином, функція **`deco()`** стає декоратором функції **`func`**`()`.

Як параметр функція-декоратор приймає посилання на функцію, поведінку якої необхідно змінити, і повинна повертати посилання на ту ж функцію або яку-небудь іншу. Наш попередній приклад еквівалентний наступному коду:

Приклад 3.276.

```
def deco(f):
    print("Тут можна писати код декоратора")
    return f
def func(x):
        print("А тепер запустили функцію func()")
    return "x = {0}".format(x)
```

```
# викликаємо функцію func() через функцію deco()
print (deco(func)(10))
```

Результат роботи:

Тут можна писати код декоратора

А тепер запустили функцію func()

x = 10

Приклад 3.275. Застосування декоратора

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped
   def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped
@makebold   #Виконується другим
@makeitalic #Виконується першим
def hello():
    return "Привіт"
print(hello())
Результат: <b><i>Привіт</i></b>
```

Приклад 3.276. Створення списку з 1 000 000 елементів

```
import time
import random
import os
import psutil
car_names =
['Audi','Toyota','Renault','Nissan','Honda','Suzuki']
colors = ['Black', 'Blue', 'Red', 'White', 'Yellow']
def car_list(cars):
    all_cars = []
```

```
    for i in range (cars):
        car = {
            'id': i,
            'name': random.choice(car_names),
            'color': random.choice(colors)
        }
        all_cars.append(car)
    return all_cars
# Вимірюємо затрати памяті
process = psutil.Process(os.getpid())
print('Memory before list is created: ' +
str(int(process.memory_info().rss/1000000)),"МБ")
# Виклик  функциії car_list та вимірювання часу
t1 = time.perf_counter()
cars = car_list(1000000)
t2 = time.perf_counter()
# Вимірюємо затрати памяті
process = psutil.Process(os.getpid())
print('Memory after list is created: ' +
str(int(process.memory_info().rss/1000000)), "МБ")
print('Took {0:.3f} seconds'.format(t2-t1))
```

Результат роботи:

Memory before list is created: 32 МБ

Memory after list is created: 317 МБ

Took 2.629 seconds

Приклад 3.277. Створення `yield`-генератора з 1000000 елементів

```
import time
import random
import os
import psutil
car_names = ['Audi', 'Toyota', 'Renault', 'Nissan',
'Honda', 'Suzuki']
colors = ['Black', 'Blue', 'Red', 'White', 'Yellow']
```

```
def car_list_gen(cars):
    for i in range (cars):
        car = {
            'id':i,
            'name':random.choice(car_names),
            'color':random.choice(colors)
        }
        yield car
# Вимірюємо затрати пам'яті
process = psutil.Process(os.getpid())
print('Memory before list is created: ' +
str(int(process.memory_info().rss/1000000))+"МБ")
# Виклик функції car_list_gen і час
t1 = time.perf_counter()
carsfunc = car_list_gen(1000000)
cars=[i for i in carsfunc]
t2 = time.perf_counter()
# Вимірюємо затрати пам'яті
process = psutil.Process(os.getpid())
print('Memory after list is created: ' +
str(int(process.memory_info().rss/1000000)), "МБ")
print('Took {0:.8f} seconds'.format(t2-t1))
```

Результат роботи:

Memory before list is created: 32МБ

Memory after list is created: 317 МБ

Took 2.53165350 seconds

Контрольні запитання до розділу 3

1. Які правила іменування змінних у мові програмування Python вам відомі?
2. Перерахуйте основні типи даних Python.
3. Змінювані та незмінювані типи даних.
4. Опишіть особливості групового присвоювання у мові Python.
5. Які конфігурації оператора розгалуження вам відомі? Наведіть способи взаємного розміщення операторів розгалуження.
6. Які оператори циклу вам відомості? Особливості застосування операторів циклу у мові Python.
7. Опишіть визначення функцій користувача.
8. Які особливості застосування функцій-генераторів?

4. БІБЛІОТЕКА PANDAS

4.1.Основні характеристики

Бібліотека Pandas дозволяє виконувати найважливіші операції у сфері Data Science. Вона забезпечує механізми аналізу великих наборів даних. Для відображення даних наборів бібліотека дозволяє маніпулювати електронними таблицями та CSV-файлами лише за допомогою нескладних конструкцій. Бібліотека Pandas є однією з найбільш затребуваних бібліотек для Python, і вона має відносно невисокий поріг навчання.

Бібліотека відіграє важливу роль у науці про добування даних. Наступні розділи присвячені огляду бібліотеки Pandas Python. Вони допоможуть сформувати у студента навички спаціаліста з Data Science та аналітика даних.

4.1.1. Інсталяція бібліотеки `Pandas`

Інсталяція бібліотеки Pandas може бути виконана різними способами залежно від того, яке програмне забезпечення використовується. У найпростішому випадку використовують пакетний менеджер **`pip`:**

```
>>> pip install pandas
```

При використанні `Linux` додатково можливо скористатися пакетним менеджером операційної системи. Наприклад, для версії `Ubuntu` :

```
>sudo apt-get install python-pandas
```

Після інсталяції необхідно перевірити, чи правильно та коректно працює бібліотека pandas. Для цього необхідно запустити інтерпретатор `Python` і ввести наступні команди.

```
>>> import pandas as pd
>>> pd.test()
```

В результаті у вікні терміналу повинен з'явитися текст, який містить приблизно таку інформацію:

```
running: pytest --skip-slow --skip-network --skip-db
C:\Anaconda3\lib\site-packages\pandas
```

```
================ test session starts ================
platform win32 -- Python 3.7.6, pytest-5.3.5, py-1.8.1,
pluggy-0.13.1
rootdir: C:\
plugins: hypothesis-5.5.4, arraydiff-0.3, astropy-
header-0.1.2, doctestplus-0.5.0, openfiles-0.4.0,
remotedata-0.3.2
collected 63152 items / 8 skipped / 63144 selected
```

Інсталяція Python агрегатором Anaconda

Це також простий спосіб отримати доступ до бібліотеки Pandas. Цей спосіб також дозволяє отримати доступ до багатьох інших важливих бібліотек, таких як NumPy та SciPy.

Для цього необхідно пройти за посиланням https://www.anaconda.com/distribution/#windows і завантажити потрібну версію дистрибутиву Anaconda. Після завантаження інсталятора потрібно виконати просту процедуру налаштування залежно від використовуваного середовища для роботи з Python.

4.1.2. Архітектура бібліотеки Pandas

Pandas Series – послідовності в Pandas можна розглядати як одновимірний масив, який використовується для обробки та маніпулювання даними, які в ньому зберігаються.

Pandas DataFrame – це структура даних у Pandas, яка складається з декількох послідовностей. Головним чином, Pandas DataFrame можна порівняти з двовимірним масивом. Вони широко використовуються для зберігання даних і маніпулювання ними.

Розглянемо коротко ієрархію файлів у Pandas.

- **pandas / core:** складається із структур даних бібліотек Pandas.
- **pandas / src:** містить основні функції Pandas, що залежать від певних алгоритмів. Зазвичай вони написані на C або Cython.

• **pandas / io:** містить інструменти для вводу та виводу, файли, дані тощо.

• **pandas / tools:** коди та алгоритми для різних функцій та операцій у Pandas. Наприклад: Merge and join, concatenation тощо.

• **pandas / util:** складається із засобів тестування та різних інших утиліт для налагодження бібліотеки.

• **pandas / rpy:** складається з інтерфейсу, який допомагає підключитися до R. Він називається R2Py.

4.1.3. Операції у бібліотеці Pandas

Розглянемо основні операції, які використовуються у Pandas.

Операція Slicing *(нарізка)*

Ви можете зробити нарізку або вирізати дані з DataFrame, щоб отримати певну необхідну вам частину даних. Це допомагає відфільтрувати важливі дані з великого набору даних.

Приклад 4.1. Нехай ми маємо дані типу **Series** під назвою “ser”, що складаються з елементів [1, 4, 6, 7, 3, 8].

Тоді за допомогою команди ser[0: 3] ми можемо зробити нарізку набору даних, щоб отримати перші три елементи [1, 4, 6].

Операції Merging and Joining (Злиття та приєднання)

Merging, тобто злиття, допомагає об’єднати кілька наборів даних. Можна навіть вибрати стовпці, які хочемо підтримувати спільними для двох наборів даних. Але злиття може працювати лише по колонці. Для приєднання по індексу ми використовуємо операцію Join.

Нехай дано набори даних A та B.

Таблиця 4.1.

Набір даних A

A	Номер елементу	Категорія	Назва
0	1	Овочі	Огірки
1	2	Овочі	Помідори
2	3	Фрукти	Яблука
3	4	Овочі	Картопля
4	5	Фрукти	Груші

Таблиця 4.2.

Набір даних B

B	Номер елементу	Ціна
0	1	50
1	2	85
2	3	24
3	4	60

Після виконання `marging` (злиття) отримуємо:

Таблиця 4.3.

Результат злиття

A	Номер елементу	Категорія	Назва	Ціна
0	1	Овочі	Огірки	50
1	2	Овочі	Помідори	85
2	3	Фрукти	Яблука	24
3	4	Овочі	Картопля	60

Операція `Concatenation` *(конкатенація)*

Ця операція склеює порядково два набори даних та в результаті утворює єдиний набір.

Нехай дано набори даних A та B.

Таблиця 4.4.

Набір A

A	Номер елементу	Категорія	Назва
0	1	Овочі	Огірки
1	2	Овочі	Помідори
2	3	Фрукти	Яблука
3	4	Овочі	Картопля
4	5	Фрукти	Груші

Таблиця 4.5.

Набір B

B	Номер елементу	Категорія	Назва
0	4	Овочі	Баклажани
1	5	Фрукти	Сливи
2	6	Фрукти	Персики
3	7	Овочі	Кабачки
4	8	Фрукти	Апельсини

Після виконання операції конкатенації отримаємо такий набір даних

Таблиця 4.6.

Результат конкатенації

A	Номер елементу	Категорія	Назва
0	1	Овочі	Огірки
1	2	Овочі	Помідори
2	3	Фрукти	Яблука
3	4	Овочі	Картопля
4	5	Фрукти	Груші
0	4	Овочі	Баклажани
1	5	Фрукти	Сливи
2	6	Фрукти	Персики
3	7	Овочі	Кабачки
4	8	Фрукти	Апельсини

Операція `Index changing` *(зміна індексу)*

Ми можемо змінити індекс будь-якого елементу даних. Це дає можливість краще маніпулювати даними.

Таблиця 4.7.

Початковий набір даних

	Номер елементу	Категорія	Назва
0	1	Овочі	Огірки
1	2	Овочі	Помідори
2	3	Фрукти	Яблука
3	4	Овочі	Картопля
4	5	Фрукти	Груші
0	4	Овочі	Баклажани
1	5	Фрукти	Сливи
2	6	Фрукти	Персики
3	7	Овочі	Кабачки
4	8	Фрукти	Апельсини

У цьому наборі даних ми можемо вибрати індексним будь-який із стовпців. Наприклад, можемо вибрати "Номер елементу" та індексувати його.

Таблиця 4.8.

Результуючий набір даних

	Номер елементу	Категорія	Назва
0	1	Овочі	Огірки
1	2	Овочі	Помідори
2	3	Фрукти	Яблука
3	4	Овочі	Картопля
4	5	Фрукти	Груші
0	4	Овочі	Баклажани
1	5	Фрукти	Сливи
2	6	Фрукти	Персики
3	7	Овочі	Кабачки
4	8	Фрукти	Апельсини

Операція `GroupBy` *(групування)*

Ця операція може використовуватися по-різному, в основному для групування даних на основі умови.

Таблиця 4.9.

Набір даних для групування

	Номер елементу	Категорія	Назва	Ціна
0	1	Овочі	Огірки	50
1	2	Овочі	Помідори	85
2	3	Фрукти	Яблка	24
3	4	Овочі	Картопля	60
4	5	Фрукти	Груші	75
0	4	Овочі	Баклажани	34
1	5	Фрукти	Сливи	40
2	6	Фрукти	Персики	89
3	7	Овочі	Кабачки	12
4	8	Фрукти	Апельсини	45

За допомогою операції `groupby` ми можемо групувати овочі та фрукти:

Таблиця 4.10.

Результат групування

Категорія	Номер елементу	Ціна
Фрукти	5.4	54.6
Овочі	3.6	48.2

Операція `Data Munging` *(Змінення даних)*

Ця операція допомагає нам перетворити дані однієї форми в іншу. **Наприклад:** Перетворення CSV в HTML.

4.1.4. Властивості `Pandas`

Бібліотека `Pandas` має велику кількість характеристик, найважливішими з яких є такі:

1. **Маніпулювання даними**: `Pandas` включає велику кількість функцій та властивостей для виконання різних видів операцій з наборами даних.

2. **Обробка пропусків у даних**: набори даних недосконалі, часто багатьох даних бракує. З цим ефективно працює бібліотека.

3. **Підтримка формату файлу**: `Pandas` підтримує різні формати файлів як для вводу, так і для виводу.

4. **Очищення даних**: Дані можуть бути дуже зашумленими. Бібліотека `Pandas` надає різноманітні інструменти, які допомагають очистити дані та зробити їх придатними для аналізу.

5. **Візуалізація**: можливо переглянути результати аналізу даних за допомогою `Pandas` візуально, що допомагає краще їх зрозуміти.

6. **Підтримка `Python`**: `Pandas` працює разом з Python. Що дає нам доступ до інших бібліотек з `Python`, таких як `NumPy`, `SciPy` та `MatPlotLib`.

4.1.5. Сфери застосування бібліотеки `Pandas`

Існують два основні напрямки застосування бібліотеки `Pandas`.

Аналіз даних

Аналіз даних – це одне з основних застосувань Pandas. Бібліотека здатна обробляти величезні набори даних, може бути використана для аналізу величезних обсягів даних. Можливості маніпуляцій даними дозволяють очищати їх та фільтрувати. Після попередньої обробки дані можуть бути проаналізовані. Деякі сфери, в яких найчастіше використовують аналіз даних за допомогою Pandas:

• **Економіка**: велика частина економіки залежить від аналізу даних та спроб знайти тенденції та подібності. Для цього широко застосовують бібліотеку Pandas.

• **Статистика**: Бібліотека Pandas включає багато функцій для виконання різних статистичних операцій.

• **Веб-аналітика**: Бібліотека Pandas допомагає прочитати та проаналізувати трафік веб-сайту, щоб надати корисну інформацію та вдосконалити веб-сайт різними способами.

Машинне навчання

Бібліотека Pandas допомагає попередньо обробити дані для моделі з метою вивчення та прогнозування результатів. Без Pandas моделі машинного навчання не змогли б ефективно читати дані.

Можливість імпортувати дані та аналізувати їх є надзвичайно важливим етапом машинного навчання.

Ця властивість бібліотеки використовується у таких випадках:

• **Рекомендації**: завдяки веб-сайтам, які використовують машинне навчання, таким як Netflix та Spotify, пропонуються цікаві рекомендації для користувачів.

• **Фінанси**: машинне навчання можна використовувати для прогнозування запасів. Pandas використовується для обробки даних попередніх угод на фондовому ринку, які допомагають передбачити майбутні угоди.

• **Обробка природної мови (NLP)**: використання машинного навчання для розуміння людської мови та її тонкощів.

Бібліотеку Pandas використовують майже всі компанії, які працюють у сфері Data Science. Деякі з них:

1. Facebook
2. IBM
3. AppNexus
4. JP Morgan Chase
5. Goldman Sachs
6. Spotify
7. Pepsico
8. AQR Capital Management
9. Vital labs
10. Uber

Pandas – це необхідна бібліотека для будь-кого, хто працює у сфері обробки даних або машинного навчання. Ці напрямки є надзвичайно прибутковими та цікавими галузями, і зараз вони швидко розвиваються. Тому вивчення Pandas є надзвичайно важливим.

4.2. Основна функціональність бібліотеки Pandas

Бібліотека Pandas популярна завдяки своїй функціональності. Базовий функціонал Pandas рекомендується добре засвоїти як початківцям, так і тим, хто вже має досвід роботи з Pandas.

Перед початком роботи з Pandas її необхідно імпортувати:

```
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916
64 bit (AMD64)] on win32
>>>import numpy as np
>>>import pandas as pd
```

Розглянемо 3 головні структури даних, які у процесі навчання будемо використовувати у бібліотеці Pandas: Andex, Series і DataFrame:

Приклад 4.2.

```
>>>dataflair_index =pd.date_range('1/1/2000', periods=8)
>>>dataflair_s1 = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
>>>dataflair_df1 = pd.DataFrame(np.random.randn(8, 3), index=dataflair_index,columns=['A', 'B', 'C'])
```

Розпочнемо з основних функціональних можливостей Pandas.

1. функція **head ()**
2. функція **tail ()**
3. Атрибути
4. Гнучкі бінарні операції

Щоб переглянути початок чи кінець довгої серії, можливим є використання функцій **head()** або **tail()**.

4.2.1. Функція head()

Створимо серію з 1000 випадкових значень:

```
>>>dataflair = pd.Series(np.random.randn(1000))
```

Переглянемо початкові дані даної серії за замовчуванням:

Приклад 4.3.

```
>>>dataflair.head()
0   -0.010748
1    1.642721
2    1.577784
3   -1.606261
4   -0.734933
dtype: float64
```

Переглянемо початкові дані даної серії з параметром:

Приклад 4.4.

```
>>>dataflair.head(10)
0   -0.010748
1    1.642721
2    1.577784
3   -1.606261
4   -0.734933
5    1.438641
```

```
6   -0.297310
7    0.300330
8   -1.118943
9    0.660390
dtype: float64
```

4.2.2. Функція `tail()`

Виводимо прикінцеві значення серії з використанням функції **`tail()`** за замовчуванням та з параметром.

Приклад 4.5.

```
>>>dataflair.tail()
995   -0.804745
996    1.616770
997    0.413478
998   -0.038675
999   -1.048558
dtype: float64
```

```
>>>dataflair.tail(10)
990    0.069612
991    1.447366
992    0.680378
993   -1.606259
994    1.494095
995   -0.804745
996    1.616770
997    0.413478
998   -0.038675
999   -1.048558
dtype: float64
```

4.2.3. Атрибути

Атрибути відіграють важливу роль у базовій функціональності бібліотеки `Pandas`, що допомагає швидко аналізувати, очищати та готувати дані. Об'єкти `Pandas` мають ряд атрибутів, що дозволяють отримати доступ до методів.

`Shape`: Цей атрибут визначає розмір по тій чи іншій осі

`Axis labels` (мітки по осях) :

1. **`Series`**: індекс (лише одна вісь)
2. **`DataFrame`**: індекс (рядки та стовпці)

Приклад 4.6.

Приклади задавання атрибутів:

```
>>>dataflair_df1[:2]
```

	A	B	C
2000-01-01	0.039691	-0.242037	-0.424762
2000-01-02	1.575138	0.940018	-0.546221

Цей код друкує два останні значення, попередньо заданого нами **DataFrame** з назвою `dataflair_df1`.

Приклад 4.7.

```
>>>dataflair_df1.columns = [x.lower() for x in
dataflair_df1.columns]
>>>dataflair_df1
```

	a	b	c
2000-01-01	0.039691	-0.242037	-0.424762
2000-01-02	1.575138	0.940018	-0.546221
2000-01-03	-2.293494	0.124177	0.152296
2000-01-04	0.632345	-0.750684	0.070756
2000-01-05	1.187279	0.488144	2.141536
2000-01-06	2.187439	0.791116	-1.086204
2000-01-07	1.616960	-0.163603	-0.213750
2000-01-08	-0.841585	-1.169087	-1.120305

За допомогою цієї функції змінюємо назви верхнього регістру на малі.

Якщо потрібно отримати лише дані, які знаходяться всередині структури даних `Pandas`, потрібно використовувати властивість `values`.

Приклад 4.8.

```
>>>dataflair_s1.values
array([-0.20148124,  0.29525041,  1.26973079, -0.40087586,
1.28779925])
>>>dataflair_df1.values
   array([[ 0.03969064, -0.24203655, -0.4247622 ],
```

```
       [ 1.57513788,  0.94001812, -0.54622148],
       [-2.29349365,  0.12417669,  0.15229627],
       [ 0.63234461, -0.75068441,  0.07075602],
       [ 1.18727911,  0.48814356,  2.14153636],
       [ 2.18743936,  0.79111617, -1.08620369],
       [ 1.61695999, -0.16360311, -0.21375049],
       [-0.84158533, -1.1690871 , -1.12030483]])
```

4.3. Гнучкі бінарні операції

У двійкових операціях між структурами даних у бібліотеці `Pandas` існують дві життєво важливі точки:

1. Передавання поведінки між об'єктами нижчої розмірності та об'єктами вищої розмірності.
2. Відсутні дані під час обчислень.

Необхідно навчитися керувати цими двома проблемами. До того ж з ними можна боротися одночасно.

4.3.1. Передавання поведінки

Для передавання поведінки **`Series`** вхід є основним. Ми можемо зіставляти індекс або стовпці за допомогою ключового слова `axis ()`.

Приклад 4.9.

```
>>>dataflair_df = pd.DataFrame({'one' :
pd.Series(np.random.randn(3), index=['a', 'b', 'c']),'two'
: pd.Series(np.random.randn(4), index=['a', 'b', 'c',
'd']),'three' : pd.Series(np.random.randn(3), index=['b',
'c', 'd'])})
>>>dataflair_df
          one          two      three
a   -1.646458   -0.997316    NaN
b   -0.114519   -1.581440   -1.953970
c    0.129243    1.120905   -0.865249
d    NaN         1.040719   -0.447942
```

Приклад 4.10.

```
>>>row = dataflair_df.iloc[1]
>>>column = dataflair_df['two']
>>>dataflair_df.sub(row, axis='columns')
        one       two     three
a -1.531939  0.584124       NaN
b  0.000000  0.000000  0.000000
c  0.243762  2.702345  1.088721
d       NaN  2.622159  1.506028
```

Приклад 4.11.

```
>>>dataflair_df.sub(column, axis='index')
 one          two      three
a -0.649142  0.0          NaN
b  1.466921  0.0    -0.372530
c -0.991662  0.0    -1.986154
d       NaN  0.0    -1.488661
>>>dataflair_df.sub(column, axis=0)
    one       two   three
a -0.649142  0.0       NaN
b  1.466921  0.0 -0.372530
c -0.991662  0.0 -1.986154
d       NaN  0.0 -1.488661
```

4.3.2. Багатоіндексований рівень `DataFrame`

Використовуючи серії, можливо вирівняти багатоіндексований рівень для `DataFrame`.

Приклад 4.12.

```
>>>dataflair_dfmi = dataflair_df.copy()
>>>dataflair_dfmi.index =
pd.MultiIndex.from_tuples([(1,'a'),(1,'b'),(1,'c'),(2,'a')]
,names=['first','second'])
```

```
>>>dataflair_dfmi.sub(column, axis=0, level='second')
dataflair_dfmi = dataflair_df.copy()
dataflair_dfmi.index =
pd.MultiIndex.from_tuples([(1,'a'),(1,'b'),(1,'c'),(2,'a')]
,names=['first','second'])
dataflair_dfmi.sub(column, axis=0, level='second')
                  one       two       three
    first second
    1     a   -0.649142    0.000000         NaN
          b        1.466921    0.000000     -0.372530
          c       -0.991662    0.000000     -1.986154
    2     a            NaN     2.038035      0.549374
```

Серії та індекс підтримують вбудовану функцію **`divmod ()`**. Вона одночасно виконує операцію **`floor`** разом з остачею від ділення та повертає кортеж з двох елементів того ж типу.

Розглянемо спочатку серії.

Приклад 4.13.

```
>>>dataflair_s = pd.Series(np.arange(10))
>>>dataflair_s
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int32
```

Виконаємо функцію `divmod()`

Приклад 4.14.

```
>>>div, rem = divmod(dataflair_s, 3) #Dividing by 3
>>>div
0    0
1    0
```

```
2    0
3    1
4    1
5    1
6    2
7    2
8    2
9    3
dtype: int32
```

Приклад 4.15.

```
>>>rem
0    0
1    1
2    2
3    0
4    1
5    2
6    0
7    1
8    2
9    0
dtype: int32
```

Приклад 4.16.

Розглянемо **index**.

```
>>> dataflair_idx = pd.Index(np.arange(10))
>>> dataflair_idx
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
```

Приклад 4.17.

```
>>> div, rem = divmod(dataflair_idx, 3)
>>> div
Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')
>>> rem
Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```

Ми також можемо зробити `divmod()` поелементно.

Приклад 4.18.

```
div,rem=divmod(dataflair_s,[2,2,3,3,4,4,5,5,6,6])  # Перший елемент буде розділений на 2, другий елемент на 3, третій на 3 тощо.
```

```
>>> div, rem = divmod(dataflair_s, [2, 2, 3, 3, 4, 4, 5, 5,
6, 6])
>>> div
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int32
>>>rem
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
9    3
dtype: int32
```

4.3.3. Відсутні дані в **Pandas**

У **DataFrame** and **Series** арифметична функція дає можливість ввести **fill_value**, яке в основному замінює значення, коли якесь значення відсутнє на своєму місці. Коли додають два об'єкти **DataFrame**, то можливо обробляти **NaN** як 0. Однак, якщо відсутні обидва **DataFrame**, це значення матиме **NaN**. Його можливо замінити його іншим значенням, використовуючи функцію **fillna** пізніше.

Приклад 4.19.

```
>>>dataflair_df
        one      two        three
a -1.646458  -0.997316      NaN
b -0.114519  -1.581440  -1.953970
c  0.129243  1.120905   -0.865249
d     NaN    1.040719   -0.447942
```

Приклад 4.20.

```
>>> dataflair_df2 = pd.DataFrame({'one' :
pd.Series(np.random.randn(3), index=['a', 'b', 'c']),'two'
: pd.Series(np.random.randn(4), index=['a', 'b', 'c',
'd']),'three' : pd.Series(np.random.randn(3), index=['b',
'c', 'd'])})

>>> dataflair_df2
        One           two     three
   a  0.341549   0.306253      NaN
   b -0.041894  -0.564515   0.824091
   c  1.627205   0.836437  -0.501329
   d     NaN     0.037271  -0.345932

>>> dataflair_df + dataflair_df2
             one       two        three
   a -1.304909  -0.691063       NaN
   b -0.156412  -2.145955  -1.129880
   c 1.756447   1.957342   -1.366579
   d     NaN    1.077990   -0.793874

>>> dataflair_df.add(dataflair_df2,fill_value=0)
#does the same thing as '+' operator
      one           two       three
   a    -1.304909    -0.691063    NaN
   b    -0.156412    -2.145955    -1.129880
   c    1.756447     1.957342     -1.366579
   d         NaN     1.077990     -0.793874
```

У цьому параграфі розглянуті основні функції поряд з деякими гнучкими порівняннями та булевими скороченнями.

4.4. Застосування методів Pandas

В цій лекції розглянемо застосування методів pipe(), apply(), applymap(). Застосування цих методів залежить від того, як обробляти дані: поелементно, по рядках чи по стовпцях.

Метод pipe() забезпечує табличну обробку даних.

Метод apply() виконує операції по рядках або стовпцях.

Метод applymap() виконує операції поелементно.

На першому кроці завантажимо бібліотеку Pandas та модуль NunPy.

```
>>> import pandas as pd
>>> import numpy as np
```

4.4.1. Застосування методу pipe()

Функція виконує операції над своїми параметрами. Такі параметри часто називають pipe-аргументами. Відповідна операція може бути виконана на усьому наборі DataFrame або Series. Застосування даної функції дозволяє спростити код виконання масових операцій над наборами даних

Розглянемо роботу методу pipe () у **Pandas**.

Спочатку сформуємо функцію зворотного виклику, яка буде виконувати операцію над кожним з елементів типу **Series**.

Приклад 4.21.

```
>>>def mult(ele1,ele2):
        return ele1*ele2
Створимо об’єкт Series та роздрукуємо його
>>>mydata_s1 = pd.Series([12,12,13,14,15])
>>>print(mydata_s1)
0    12
1    12
2    13
3    14
4    15
dtype: int64
```

Застосуємо метод pipe() до об'єкта **Series**.

```
>>>mydata_s1.pipe(mult,2)
0    24
1    24
2    26
3    28
4    30
dtype: int64
```

Ми бачимо, що кожен елемент об'єкта mydata_s1 збільшився шляхом множення на другий параметр методу pipe().

Для застосування методу pipe() до **DataFrame** створимо об'єкт:

Приклад 4.22.

```
>>>def difer(ele1,ele2):
            return ele1-ele2
>>>mydata_df2=pd.DataFrame(5*np.random.randn(5,3),columns=[
'n1','n2','n3'])
>>>mydata_df2
          n1         n2       n3
0 3.496179   -0.200216  3.615396
1 1.889404   -0.867754  2.618930
2 1.190007   3.704108   -6.517730
3 2.042387   2.970400   8.976790
4 2.066436   -10.685903       -2.583032
```

До даного об'єкта застосуємо метод pipe():

```
>>>mydata_df2.pipe(difer,3)
      n1             n2            n3
0  0.496179       -3.200216       0.615396
1  -1.110596      -3.867754       -0.381070
2  -1.809993      0.704108        -9.517730
3  -0.957613      -0.029600        5.976790
4  -0.933564      -13.685903     -5.583032
```

4.4.2. Застосування методу **aply()**

Формат методу:

```
DataFrame.apply(func, [axis=0], [ **kwds])
```

Метод `apply()` можна застосувати довільно до осей **DataFrame** або до **Series**. За замовчуванням операція виконується по стовпцю, приймаючи кожен стовпець як масив. Дії, які застосовують до кожного елемента стовпця, визначають в попередньо заданій функції зворотного виклику. Користувач передає посилання на функцію, а метод застосовує її до всіх значень **DataFrame** або **Series**.

Визначимо функцію зворотного виклику для `apply ()`:

Приклад 4.23.

```
>>>def formul(el1,el2,el3):
    return (el1+el2)*el3
```

Застосуємо цей метод спочатку до об'єкта Series:

```
>>>mydata_s1 = pd.Series([2,3,4,6,6])
>>>print(mydata_s1)
>>>mydata_s1.apply(formul,el2=3,el3=2)
0    2
1    3
2    4
3    6
4    6
dtype: int64


0    10
1    12
2    14
3    18
4    18
dtype: int64
```

При застосуванні методу `apply()` до **DataFrame** можливо вибирати індекс осі. Створимо функцію зворотного виклику `sum()` та набір даних **DataFrame**:

Приклад 4.24.

```
>>>def sum(el1,el2):
     return el1+el2
```

```
>>>mydata_df2=pd.DataFrame(np.random.randn(4,3),columns=['n
1','n2','n3'])
>>>mydata_df2
          n1        n2           n3
    0   -0.206522   1.404361    0.869129
    1   -0.454444   1.278791   -0.746351
    2   1.192815    0.780912   -0.906040
      3 -0.560947   1.498817    0.159539
>>>mydata_df2.apply(suma,el2=1)
              n1          n2          n3
    0   2.463387   -0.315078   1.933329
    1   -0.079420   0.982241   1.624842
    2   0.619850    0.201328   1.904604
    3   0.792562    0.096900   1.557962
```

Використовуючи вбудовану функцію NumPy, можемо отримати суми по осях

```
>>>mydata_df2.apply(np.sum,axis=0)
n1   -0.203621
n2   -3.034608
n3    3.020736
dtype: float64
>>>mydata_df2.apply(np.sum,axis=1)
0    1.081638
1   -0.472337
2   -0.274218
3   -0.552576
dtype: float64
```

4.4.3. Застосування методу `aplymap()`

Цей метод дозволяє застосувати поелементно функцію зворотного виклику. У цьому випадку можемо застосовувати лише функцію, яка приймає скаляр та повертає скаляр для кожного елемента DataFrame.

Приклад 4.25.

```
>>>mydf = pd.DataFrame([[1, 2, 3], [44, 55, 66], [777, 888,
999]])

>>>mydf
     0    1    2
0  1      2    3
1    44   55   66
2    777  888  999
mydf.applymap(lambda x: len(str(x)))
     0    1    2
0    1    1    1
1    2    2    2
2    3    3    3
```

При поелементній обробці методом `applymap()` значення `NA` ігноруються у версії `pandas 1.x`.

Приклад 4.26.

```
mydf.iloc[1,1] = pd.NA
mydf
     0    1     2
0    1    2     3
1    44   <NA>  66
2    777  888   999

>>>mydf.applymap(lambda x: len(str(x)), na_action='ignore')
   0     1    2
0    1    1    1
1    2    <NA> 2
2    3    3    3
```

Контрольні запитання до розділу 4

1. Яке призначення бібліотеки `Pandas`? Поясніть процес установки бібліотеки `Pandas`.
2. Перерахуйте операції, які дозволяє виконувати бібліотека `Pandas`.
3. Які найважливіші властивості має бібліотека `Pandas`?
4. Які сфери застосування бібліотеки `Pandas`?
5. Основні функції і атрибути бібліотеки `Pandas`.
6. Функції для роботи з структурою даних `Series`.
7. Властивості і функції структури даних `DataFrame`.

5. ОСНОВНІ ПРИНЦИПИ РОБОТИ

NUMPY, SCIPY, MATPLOTLIB

Numpy – це основна бібліотека наукових обчислень на Python. Вона використовує як базовий об'єкт високоефективний багатовимірний масив та інструменти для роботи з цим масивом.

Масиви

Numpy масив – це сітка величин одного типу, які індексуються кортежем невід'ємних цілих чисел. Величину розмірності масиву називають рангом масиву; форма масиву – це кортеж цілих чисел, які задають розмір масиву вздовж кожної розмірності.

Створення масиву

Існує 5 загальних механізмів створення масивів. Розглянемо два основні.

1. Перетворення з інших структур Python (наприклад, списки, кортежі).

2. Внутрішні об'єкти створення масиву numpy (наприклад, arange, ones, zeros тощо).

5.1. Перетворення об'єктів Python в NumPy масиви

Можливо ініціалізувати numpy масиви із вкладених списків Python та отримати доступ до елементів за допомогою квадратних дужок:

Приклад 5.1.

```
import numpy as np
print('Створення одновимірного масиву')
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class
'numpy.ndarray'>"
print('Розмір')
print(a.shape)            # Prints "(3,)"
print('Друк елементів')
print(a[0], a[1], a[2])   # Prints "1 2 3"
```

```
print('Модифікація a[0]')
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"
print('Створення двовимірного масиву')
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print('Розмір')
print(b.shape)                     # Prints "(2, 3)"
print('Друк елементів')
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print('Розмір')
print(b.shape)                     # Prints "(2, 3)"
print('Створення тривимірного масиву')
b = np.array([[1,2,3,1],[4,5,6,1],[7,8,9,10]])    # Create
a rank 3 array
print('Розмір')
print(b.shape)                     # Prints "(3, 4)"
```

Результат:

```
Створення одновимірного масиву
<class 'numpy.ndarray'>
Розмір
(3,)
Друк елементів
1 2 3
Модифікація a[0]
[5 2 3]
Створення двовимірного масиву
Розмір
(2, 3)
Друк елементів
```

```
1 2 4
Розмір
(2, 3)
Створення тривимірного масиву
Розмір
(3, 4)
```

5.1.1. Індексація масиву

Зрізи: Подібно до списків Python, в масивах можна використовувати зрізи. Оскільки масиви можуть бути багатовимірними, потрібно вказати фрагмент для кожного виміру масиву:

Приклад 5.2.

```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3]
print(a)
print(b)
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as
a[0, 1]
print(a[0, 1])   # Prints "77"
```

Результат:

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[2 3]
 [6 7]]
2
77
```

Можна змішати цілочисельну індексацію з індексуванням фрагментів. Це дозволить отримати масив нижчого рангу, ніж початковий масив.

Приклад 5.3.

```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
# We can make the same distinction when accessing columns
of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape) # Prints "[[ 2]
                            #          [ 6]
                            #          [10]] (3, 1)"
```

Результат:

[5 6 7 8] (4,)

[[5 6 7 8]] (1, 4)

[2 6 10] (3,)

[[2]

 [6]

 [10]] (3, 1)

5.1.2. Індексація масиву цілих чисел

Якщо відбуваться індексація масиву за допомогою зрізу, результат задавання масиву завжди буде підмасивом початкового масиву.

Навпаки, індексація цілого масиву дозволяє будувати довільні масиви, використовуючи дані з іншого масиву.

Приклад 5.4.

```
    a = np.array([[1,2], [3, 4], [5, 6]])
# An example of integer array indexing.
```

```
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
#Red- index of the rank1. Blue-index of the rank 2
# The above example of integer array indexing is equivalent
to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1
4 5]"
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"
#Red- index of the rank1. Blue-index of the rank 2
# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

Результат:

[1 4 5]

[1 4 5]

[2 2]

[2 2]

Один корисний трюк з цілочисельним індексуванням масиву – це вибір або виключення або зміна значення одного елемента з кожного рядка матриці:

Приклад 5.5.

```
import numpy as np
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"
# Create an array of indices
```

```
b = np.array([0, 2, 0, 1])
# Select one element from each row of a using the indices
in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
# Mutate one element from each row of a using the indices
in b
a[np.arange(4), b] += 10
print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

Результат:

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 1  6  7 11]
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Індексація булевого масиву. Індексування булевого масиву дозволяє вибирати довільні елементи масиву. Часто цей тип індексації використовується для вибору елементів масиву, які відповідають певній умові.

Приклад 5.6.

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)
# Find the elements of a that are bigger than 2;
# this returns a numpy array of Booleans of the same
```

```
# shape as a, where each slot of bool_idx tells
# whether that element of a is > 2.
print(bool_idx)       # Prints "[[False False]
                      #          [ True  True]
                      #          [ True  True]]"
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True
values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"
# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Результат:

[[False False]
[True True]
[True True]]
[3 4 5 6]
[3 4 5 6]

5.1.3.Типи даних

NumPy – це сітка елементів одного типу. NumPy пропонує великий набір числових типів даних, які можна використовувати для побудови масивів. Елементи NumPy отримують тип даних при створенні масиву, але функції, що будують масиви, зазвичай також включають необов'язковий аргумент, щоб чітко вказати тип даних.

Приклад 5.7.

```
import numpy as np
x = np.array([1, 2]) # Let numpy choose the datatype
print(x.dtype)       # Prints "int64"
x = np.array([1.0, 2.0])   # Let numpy choose the datatype
```

```
print(x.dtype)              # Prints "float64"
x = np.array([1, 2], dtype=np.int64)
    # Force a particular datatype
print(x.dtype)
    Prints "int64"
```

Результат:

int32

float64

int64

5.1.4. Математика для масивів

Основні математичні функції працюють на масивах і доступні у вигляді перезавантаження оператора, або у вигляді функцій модуля NumPy:

Приклад 5.8.

```
import numpy as np
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))
# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
```

```
print(np.multiply(x, y))
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.s
```

Результат:

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Як видно з прикладу, маємо поелементне множення, а не матричне множення. Використовуємо `dot` функцію для обчислення скалярних добутків векторів, множення вектора на матрицю та множення матриць. `Dot` доступна і як функція в модулі `NumPy`, і як метод `array`-об'єктів.

Приклад 5.9.

```
import numpy as np
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])
# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
# Matrix / vector product; both produce the rank 1 array
[29 67]
print(x.dot(v))
print(np.dot(x, v))
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
```

```
print(x / y)
print(np.divide(x, y))
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Результат:

219

219

[29 67]

[29 67]

[[19 22]

[43 50]]

[[19 22]

[43 50]]

[[5 12]

[21 32]]

[[5 12]

[21 32]]

[[0.2 0.33333333]

[0.42857143 0.5]]

[[0.2 0.33333333]

[0.42857143 0.5]]

[[1. 1.41421356]

[1.73205081 2.]]

NumPy забезпечує велику кількість корисних функцій для виконання обчислень на масивах; однією з найкорисніших є sum:

Приклад 5.10.

```
import numpy as np
x = np.array([[1,2],[3,4]])
```

```
print(np.sum(x))# Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))# Compute sum of each column;
prints "[4 6]"
print(np.sum(x, axis=1))#Compute sum of each row; prints
"[3 7]"
```

Результат:

10

[4 6]

[3 7]

Повний список математичних функцій, наданих NumPy, можна знайти в документації.

Крім обчислення математичних функцій за допомогою масивів, часто потрібно змінювати розмірність або іншим чином маніпулювати даними в масивах. Найпростіший приклад цього типу операцій – транспонування матриці; щоб транспонувати матрицю, необхідно використати атрибут T об'єкта масиву:

Приклад 5.11.

```
import numpy as np
x = np.array([[1,2], [3,4]])
print(x)     # Prints "[[1 2]
             #          [3 4]]"
print(x.T)   # Prints "[[1 3]
             #          [2 4]]"
# Note that taking the transpose of a rank 1 array does
nothing:
v = np.array([1,2,3])
print(v)     # Prints "[1 2 3]"
print(v.T)   # Prints "[1 2 3]"
```

Результат:

[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
[1 2 3]
[1 2 3]

5.1.5. Трансляція (Broadcasting)

Broadcasting – це потужний механізм, який дозволяє NumPy працювати з масивами різної форми при виконанні арифметичних операцій. Часто є менший масив і більший масив, і потрібно використовувати менший масив кілька разів, щоб виконати деяку операцію над більшим масивом.

Наприклад, припустимо, що потрібно додати постійний вектор до кожного рядка матриці. Можливо зробити це так:

Приклад 5.12.

```
mport numpy as np
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)
# Create an empty matrix with the same shape as x
# Add the vector v to each row of the matrix x with an
explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v
# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

Результат:

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Проте, у випадку, коли матриця x дуже велика, обчислення явного циклу в Python може бути повільним. Зауважимо, що додавання вектора v до кожного рядка матриці x еквівалентно формуванню матриці vv шляхом складання декількох копій v вертикально, а потім виконання елементарного додавання x та vv. Можлива така реалізація цього підходу:

Приклад 5.13.

```
import numpy as np
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))
# Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

Результат:

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

NumPy трансляція дозволяє виконувати ці обчислення, не створюючи фактично декілька копій v. Розглянемо версію коду, використовуючи трансляцію:

Приклад 5.14.

```
import numpy as np
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Результат:

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Рядок y=x+v спрацює завдяки трансляції, навіть якщо x має розмірність (4, 3), а v має розмірність (3,) ; цей рядок спрацьовує так, ніби v фактично має форму (4, 3), де кожен рядок є копією v, а сума виконується елементарно.

Трансляція двох масивів разом відповідає таким правилам:

1. Якщо масиви не мають однакового рангу, необхідно збільшувати розмір масиву меншого рангу на одиницю, поки обидва розміри за даною розмірністю не зрівняються.

2. Два масиви будемо називати сумісними за розмірністю, якщо вони мають однаковий розмір за даною розмірністю, або якщо один із масивів має розмір 1 у цій розмірності.

3. Масиви можна транслювати разом, якщо вони сумісні в усіх вимірах.

4. Після транслювання кожен масив поводиться так, ніби має розмірність, яка дорівнює поелементному максимуму елементів двох розмірностей вхідних масивів.

5. У будь-якому вимірі, де один масив мав розмір 1, а інший масив мав розмір більший за 1, перший масив поводиться так, ніби він був скопійований уздовж цього виміру.

Для більш детального пояснення зверніться до документації.

Функції, що підтримують трансляцію, відомі як універсальні функції. Перелік усіх універсальних функцій розміщений в документації.

Наведемо кілька програм трансляції:

Приклад 5.15.

```
import numpy as np
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a
column
# vector of shape (3, 1); we can then broadcast it against
w to yield
# an output of shape (3, 2), which is the outer product of
v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast
to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
```

```
# If we transpose x then it has shape (3, 2) and can be
broadcast
# against w to yield a result of shape (3, 2); transposing
this result
# yields the final result of shape (2, 3) which is the
matrix x with
# the vector w added to each column. Gives the following
matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of
shape (2, 1);
# we can then broadcast it directly against x to produce
the same
# output.
print(x + np.reshape(w, (2, 1)))
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of
shape ();
# these can be broadcast together to shape (2, 3),
producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

Результат:

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[2 4 6]
 [5 7 9]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 2  4  6]
 [ 8 10 12]]
```

Трансляція зазвичай робить код стислим і швидшим, тому необхідно прагнути використовувати її там, де це можливо.

5.1.6. Внутрішнє створення масиву NumPy¶

NumPy має вбудовані функції для створення масивів заданої форми з нульовими значеннями.

Найчастіше використовується тип float64.

Приклад 5.16.

```
import numpy as np
a=np.zeros((2, 3))
print(a)
```

Результат:

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

ones(shape) створює масив, заповнений одиничними значеннями.

Приклад 5.17.

```
import numpy as np
a=np.ones((3, 3))
print(a)
```

Результат:

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Приклад 5.18.

```
import numpy as np
c = np.full((2,2), 7)  # Create a constant array
print(c)
```

Результат:

```
[[7 7]
 [7 7]]
```

Приклад 5.19.

```
import numpy as np
d = np.eye(3)          # Create a 3x3 identity matrix
print(d)
```

Результат:

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

Приклад 5.20.

```
import numpy as np
e = np.random.random((3,3))
# Create an array filled with random values
print(e)
```

Результат:

[[0.64090337 0.50231634 0.69483244]
 [0.73571575 0.59788823 0.92113858]
 [0.16406592 0.46649887 0.13579729]]

arange () створює масиви зі значеннями, що регулярно збільшуються. Перевірте в документації повну інформацію про різні способи його використання.

Наведемо кілька прикладів:

Приклад 5.21.

```
import numpy as np
a=np.arange(10)
print(a)
```

Результат:

[0 1 2 3 4 5 6 7 8 9]

Приклад 5.22.

```
import numpy as np
a=np.arange(2, 10, dtype=float)
print(a)
```

Результат:

[2. 3. 4. 5. 6. 7. 8. 9.]

Приклад 5.23.

```
import numpy as np
a=np.arange(2, 3, 0.1)
print(a)
```

Результат:

[2. 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9]

linspace () створює масиви із заданою кількістю елементів (параметр 3), що лежать між початковим (параметр 1) та кінцевим (параметр 2) значеннями.

Приклад 5.24.

```
import numpy as np
a=np.linspace(1., 4., 6)
b=np.linspace(2., 10., 3)
print(a)
print(b)
```

Результат:

[1. 1.6 2.2 2.8 3.4 4.]

[2. 6. 10.]

indices() створює набір масивів, які об'єднані в один масив з високою розмірністю, по одному на розмір, кожен з яких представляє варіацію в цьому вимірі. Приклад ілюструє набагато краще, ніж словесний опис:

Приклад 5.25.

```
import numpy as np
a=np.indices((3,3))
b=np.indices((3,4))
print(a)
print(b)
```

```
[[[0 0 0]
  [1 1 1]
  [2 2 2]]
```

```
[[0 1 2]
 [0 1 2]
 [0 1 2]]]

[[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]]

 [[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]]
```

5.2. Бібліотека SciPy

NumPy підтримує високоефективні багатовимірні масиви та основні інструменти для обчислення та маніпулювання цими масивами. SciPy ґрунтується на цьому і забезпечує велику кількість функцій, які працюють на NumPy масивах і є корисними для різних типів наукових та інженерних застосувань.

Найкращий спосіб ознайомитися з SciPy – це перегляд документації. Виділимо деякі частини SciPy, які можуть бути корисними для нас.

5.2.1. Операції з зображенням

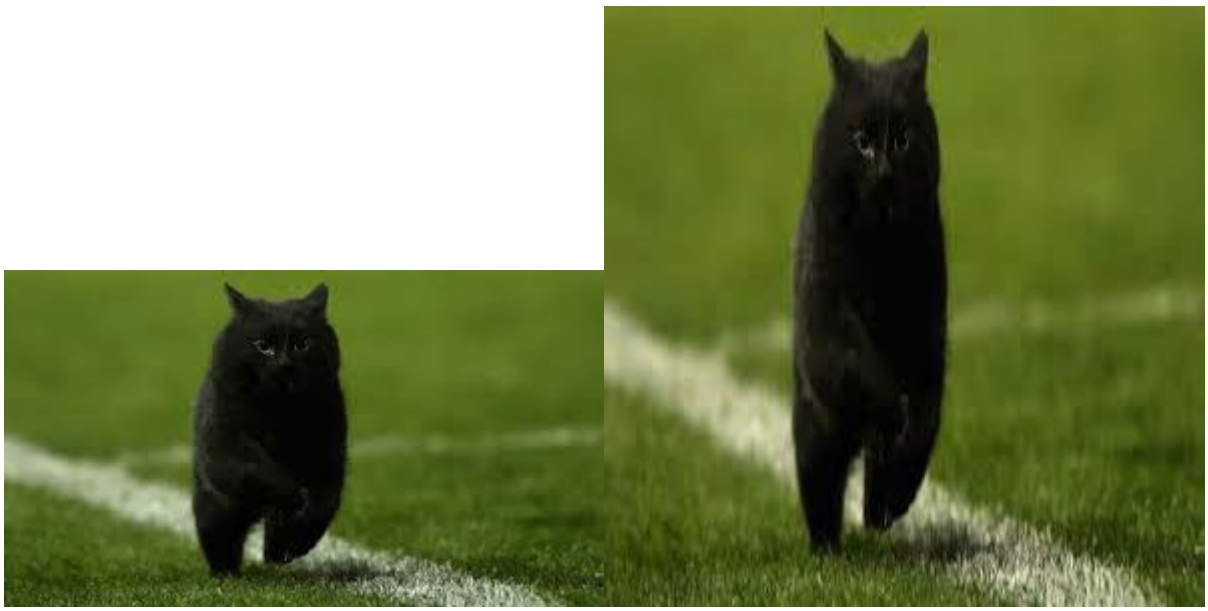
SciPy надає деякі основні функції для роботи з зображеннями. SciPy містить функції для зчитування зображень з диска в NumPy масиви, для запису масивів на диск у вигляді зображень та для зміни розмірів зображень. Ось простий приклад, який демонструє ці функції:

Приклад 5.26.

```
from scipy.misc import imread, imsave, imresize
# Read an JPEG image into a numpy array
img = imread('cat.jpg')
print(img.dtype, img.shape)
# Prints "uint8 (400, 248, 3)"
# We can tint the image by scaling each of the color
```

```
channels
# by a different scalar constant. The image has shape (400,
248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel
unchanged,
# and multiplies the green and blue channels by 0.95 and
0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]
# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))
# Write the tinted image back to disk
imsave('cat_new.jpg', img_tinted)
```

Результат:



5.3. Бібліотека `MatPlotLib`

`MatPlotLib` – це бібліотека графіків. У цьому розділі коротко ознайомимось з модулем `matplotlib.pyplot`, який забезпечує побудову графіків.

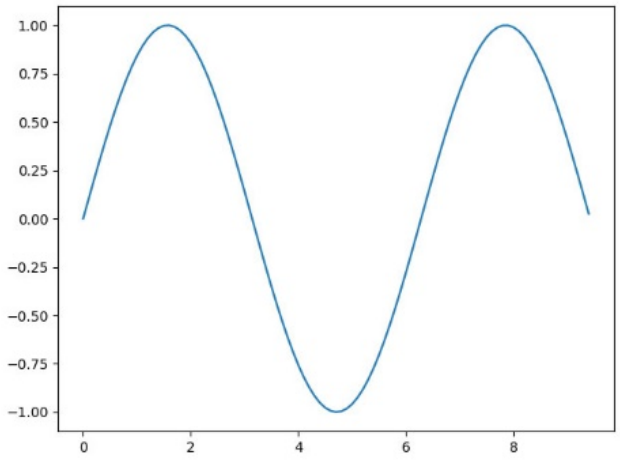
Складання графіків

Найважливішою функцією в `matplotlib` є `plot()`, яка дозволяє побудувати 2D дані. Ось простий приклад:

Приклад 5.27.

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on a sine
curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
# Plot the points using matplotlib
plt.plot(x, y)
# You must call plt.show() to make graphics appear.
    plt.show()
```

Результат:



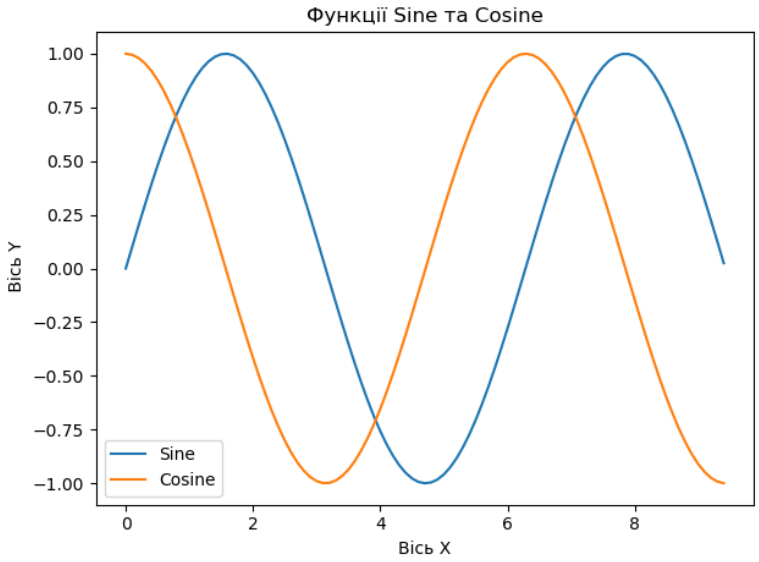
Після незначних модифікацій можна легко побудувати кілька рядків та додати назву, легенду та мітки осі:

Приклад 5.28.

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on sine and
cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
```

```
y_cos = np.cos(x)
# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('Вісь X')
plt.ylabel('Вісь Y')
plt.title('Функції Sine nf Cosine')
plt.legend(['Sine', 'Cosine'])
plt.sh
```

Результат:



5.3.1. Субграфіки

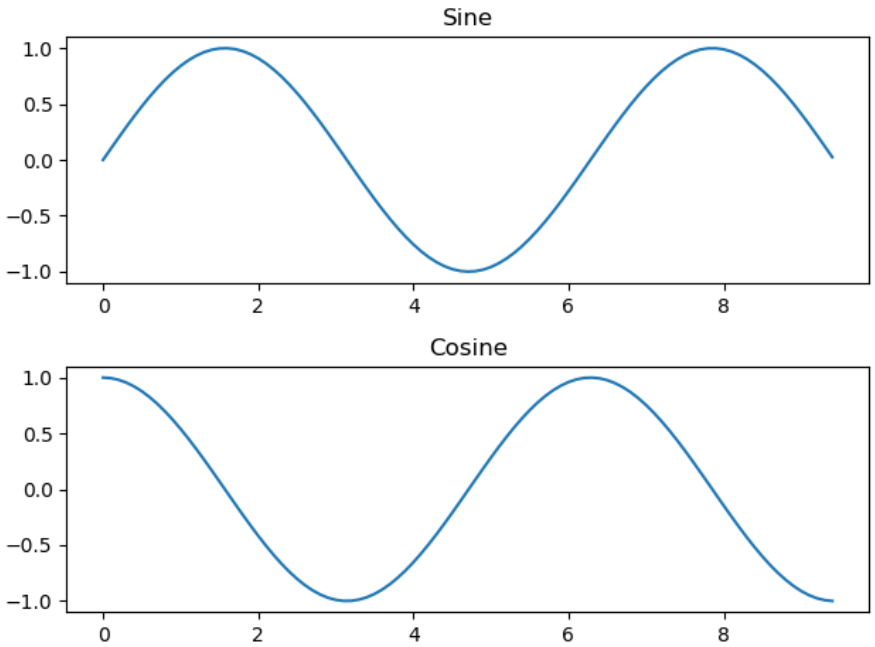
Існує можливість побудувати на одній фігурі різні графіки за допомогою функції `subplot`.

Приклад 5.29.

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on sine and
cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
```

```
# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)
# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')
# Set the second subplot as active, and make the second
plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')
# Show the figure.
plt.show()
```

Результат:



5.3.2. Зображення

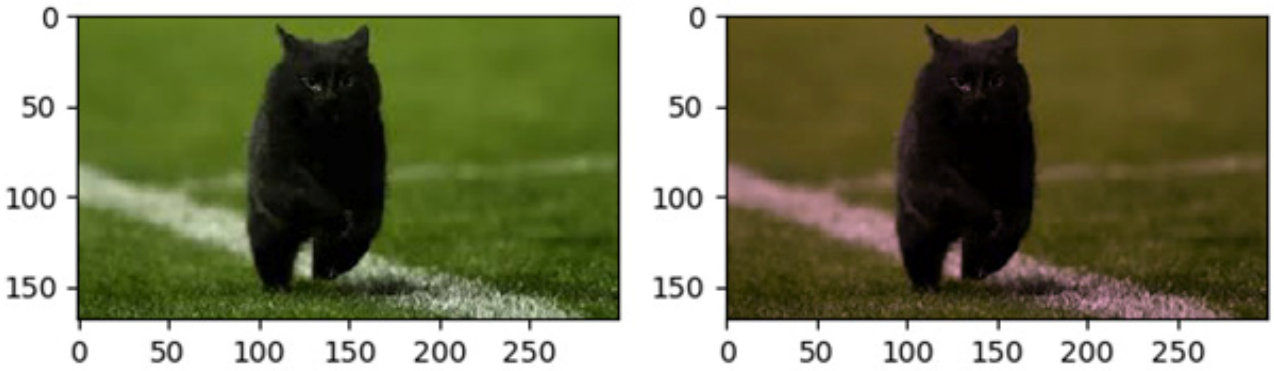
Можна використовувати функцію `imshow` для демонстрації зображень.

Приклад 5.30.

```
import numpy as np
import imageio
import matplotlib.pyplot as plt
img = imageio.imread('cat.jpg')
```

```
img_tinted = img * [1, 0.7, 0.9]
# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)
# Show the tinted image
plt.subplot(1, 2, 2)
# A slight gotcha with imshow is that it might give strange
results
# if presented with data that is not uint8. To work around
this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```

Результат:



Контрольні запитання до розділу 5

1. Опишіть принципи індексації масивів чисел в бібліотеці Pandas.
2. Які функції модуля NumPy використовуються в Pandas?
3. Дайте визначення механізму трансляції. Наведіть приклади програм трансляції.
4. Опишіть способи внутрішнього створення масиву NumPy.
5. Бібліотека ScyPy і її основні способи використання.
6. Бібліотека MatPlotLib. Застосування бібліотеки для побудови графіків, субграфіків і зображень.

6. СТРУКТУРИ ДАНИХ У PANDAS

6.1. Структура даних Series

Для того, щоб почати працювати зі структурами даних в Pandas, потрібно попередньо імпортувати необхідні модулі. Переконайтеся, що потрібні модулі встановлені на вашому комп'ютері.

Крім власне pandas знадобиться бібліотека numpy. Наші експерименти будемо проводити з використанням пакета Anaconda, як середовище розробки візьмемо PyCharm, який інтегрували з Anaconda. Він в першу чергу має цінність тому, що в ньому є редактор початкового коду, на випадок, якщо знадобиться написати досить велику програму, і інтерпретатор для швидких експериментів.

Приклад 6.1. Імпортування бібліотек

```
import numpy as np
import pandas as pd
```

Створити структуру Series можна на базі різних типів даних:

словники Python;

списки Python;

масиви з numpy: ndarray;

скалярні величини.

Конструктор класу Series має такий вигляд:

Приклад 6.2.

```
pandas.Series (data = None, index = None, dtype = None, name = None, copy = False, fastpath = False)
```

data – масив, словник чи скалярне значення, на базі якого буде побудований **Series**;

index – список індексів, який буде використовуватися для доступу до елементів **Series**. Довжина списку має дорівнювати довжині data;

dtype – об'єкт numpy.dtype, що визначає тип даних;

copy – створює копію масиву даних, якщо параметр дорівнює True, в іншому випадку нічого не робить.

У більшості випадків, при створенні **Series**, використовують тільки перші два параметра. Розглянемо різні варіанти створення структур даних.

6.1.1. Створення Series із списку Python

Найпростіший спосіб створити Series – це передати як єдиний параметр у конструктор класу список Python.

Приклад 6.3.

```
import pandas as pd
slt = pd.Series([1, 2, 3, 4, 5,7,8,9])
print(slt)
```

Результат:

```
0    1
1    2
2    3
3    4
4    5
5    7
6    8
7    9
dtype: int64
```

У даному прикладі була створена структура Series на базі списку з мови Python. Для доступу до елементів Series, в даному випадку, можна використовувати тільки додатні цілі числа – лівий стовпець чисел, що починається з нуля – це індекси елементів структури, які представлені в правій колонці.

Задамо Series у вигляді словника в pandas. Для цього передамо як другий елемент список рядків (в нашому випадку – це окремі символи). Такий крок дозволить звертатися до елементів структури Series не тільки за індексом, а й за міткою, що робить роботу з таким об'єктом схожою на роботу зі словником.

Приклад 6.4.

```
import pandas as pd
sdc=pd.Series([1,2,3,4,5,6], ['a','b','c','d','e','f'])
print(sdc)
```

Результат:

```
a    1
b    2
c    3
d    4
e    5
f    6
dtype: int64
```

Лівий стовпець містить мітки, які передані як index параметр при створенні структури. Правий стовпчик – це елементи нашої структури (data).

6.1.2. Створення Series із словника (dict)

Ще один спосіб створити структуру Series – це використовувати словник для одночасного задавання міток і значень.

Приклад 6.5.

```
import pandas as pd
md = {'a':11, 'b':22, 'c':33, 'd':44}
sdm = pd.Series(md)
print(sdm)
```

Результат:

```
a    11
b    22
c    33
d    44
dtype: int64
```

Ключі (keys) із словника md стануть мітками структури sdm, а значення (values) словника – даними (data) в структурі.

Створення **Series** *з використанням константного заповнення*

Розглянемо ще один спосіб створення структури. На цей раз значення в елементах структури будуть однаковими.

Приклад 6.6.

```
import pandas as pd
a = 7
scon = pd.Series (a, ['a', 'b', 'c'])
print (scon)
```

Результат:

```
a  7
b  7
c  7
dtype: int64
```

У створеній структурі Series є три елементи з однаковим значенням.

6.1.3. Робота з елементами Series

До елементів Series можна звертатися за індексом, при такому підході робота зі структурою не відрізняється від роботи зі списками в Python.

Приклад 6.7.

```
import pandas as pd
sd=pd.Series([1, 2, 3, 4, 5], [ 'a', 'b', 'c', 'd', 'e'])
print(sd[3])
```

Результат:

4

Можна використовувати мітки, тоді робота з Series буде схожа на роботу з словником (dict) у Python.

Приклад 6.8.

```
import pandas as pd
sd=pd.Series([1, 2, 3, 4, 5], [ 'a', 'b', 'c', 'd', 'e'])
print(sd['b'])
```

Результат:

2

Можна використовувати зрізи.

Приклад 6.9.

```
import pandas as pd
sd=pd.Series ([1, 2, 3, 4, 5], [ 'a', 'b', 'c', 'd', 'e'])
print(sd[:3])
```

Результат:

```
a    1
b    2
c    3
dtype: int64
```

У поле для індексу можна помістити умовний вираз.

Приклад 6.10.

```
import pandas as pd
sd=pd.Series([1, 2, 3, 4, 5], [ 'a', 'b', 'c', 'd', 'e'])
print(sd[sd>=3])
```

Результат:

```
c    3
d    4
e    5
dtype: int64
```

З структурами `Series` можна працювати як з векторами: додавати, множити вектор на число і т. д.

Приклад 6.11.

```
import pandas as pd
sd2=pd.Series([10,20,30,40,50], ['a','b','c','d','e'])
sd1=pd.Series([1,2,3,4,5], ['a','b','c','d','e'])
print('додавання')
print(sd1 + sd2)
print('множення')
print(sd1 * 3)
```

Результат:

додавання:

```
a    11
b    22
c    33
d    44
e    55
dtype: int64
```

множення:

```
a     3
b     6
c     9
d    12
e    15
dtype: int64
```

6.2. Структура даних DataFrame

DataFrame – це двовимірна структура – повноцінна таблиця з великою кількістю рядків і стовпців.

Перед роботою з DataFrame необхідно імпортувати бібліотеку pandas.

Конструктор класу DataFrame має такий вигляд:

Приклад 6.12.

```
class pandas.DataFrame (data = None, index = None, columns
= None, dtype = None, copy = False)
```

data – масив ndarray, словник (dict) або інший DataFrame;

index – список міток для записів (імена рядків таблиці);

columns – список міток для полів (імена стовпців таблиці);

dtype – об'єкт numpy.dtype, що визначає тип даних;

copy – створює копію масиву даних, якщо параметр дорівнює True, в іншому випадку нічого не робить.

Структуру DataFrame можна створити на базі:

словника (dict), елементами якого мають бути:

одновимірні ndarray, списки, інші словники, структури Series;

двовимірні ndarray;

структури Series;

структуровані ndarray;

інші DataFrame.

Розглянемо на практиці різні підходи до створення DataFrame.

6.2.1.Створення DataFrame із словника

В даному випадку використовується одновимірний словник, елементами якого є списки, структури Series і т. д. Почнемо з Series.

Приклад 6.13.

```
import pandas as pd
d = {"price": pd.Series ([1, 2, 3], index = [ 'v1', 'v2',
```

```
'v3']),"count": pd.Series ([10, 12, 7], index = [ 'v1',
'v2', 'v3'])}
df1 = pd.DataFrame (d)
print (df1)
print (df1.index)
print (df1.columns)
```

Результат:

```
  price  count
v1      1     10
v2      2     12
v3      3      7
Index(['v1', 'v2', 'v3'], dtype='object')
Index(['price', 'count'], dtype='object')
```

Тепер побудуємо аналогічний словник, але на елементах `ndarray`.

Приклад 6.14.

```
import pandas as pd
import numpy as np
ds = {"price":np.array([10, 20, 30]), "count":
np.array([10, 32, 45])}
dn = pd.DataFrame(ds, index=['v1', 'v2', 'v3'])
print(dn)
print(dn.index)
print(dn.columns)
```

Результат:

```
    price  count
v1     10     10
v2     20     32
v3     30     45
Index(['v1', 'v2', 'v3'], dtype='object')
Index(['price', 'count'], dtype='object')
```

Як видно – результат аналогічний попередньому. Замість ndarray можна використовувати звичайний список з Python.

6.2.2. Створення DataFrame із списку словників

До цього часу було розглянуто створення DataFrame з словника, елементи якого – структури Series, списки і масиви. Можна створювати DataFrame із списку, елементами якого є словники.

Приклад 6.15.

```
import pandas as pd
dd = [{"price": 4, "count":5}, {"price": 6, "count": 21}]
dfd = pd.DataFrame(dd)
print(dfd)
print(dfd.info())
```

```
   count  price
0      5      4
1     21      6
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 2 columns):
count    2 non-null int64
price    2 non-null int64
dtypes: int64(2)
memory usage: 112.0 bytes
None
```

6.2.3. Створення DataFrame із двовимірного масиву

Створити DataFrame можна також і з двовимірного масиву, в прикладі це буде ndarray з бібліотеки numpy.

Приклад 6.16.

```
import pandas as pd
import numpy as np
```

```
ndam = np.array([[1, 2, 3], [10, 20, 30]])
dfd = pd.DataFrame(ndam)
print(dfd)
```

Результат:

```
   0  1  2
0  1  2  3
1 10 20 30
```

6.2.4. Робота з елементами `DataFrame`

Основні підходи представлені в таблиці нижче.

Операція	Синтаксис	Повертається результат
Вибір стовпчика	`df [col]`	`Series`
Вибір рядка по мітці	`df.loc [label]`	`Series`
Вибір рядка за індексом	`df.iloc [loc]`	`Series`
Зріз по рядках	`df [0: 4]`	`DataFrame`

Розглянемо використання даних операцій на практиці. Для початку створимо `DataFrame`.

Приклад 6.17.

```
import pandas as pd
import numpy as np
dm={"price":np.array([1,2,3]),"count":np.array([10,20,30])}
dfm = pd.DataFrame(dm, index=['a', 'b', 'c'])
print(dfm)
```

Результат:

```
  price  count
a     1     10
b     2     20
c     3     30
```

Операція: вибір стовпця

Приклад 6.18.

```
import pandas as pd
import numpy as np
dm = {"price":np.array([1, 2, 3]), "count": np.array([10,
20, 30])}
dfm = pd.DataFrame(dm, index=['a', 'b', 'c'])
print(dfm['count'])
```

Результат:

a 10

b 20

c 30

Name: count, dtype: int32

Операція: вибір рядка по мітці

Приклад 6.19.

```
import pandas as pd
import numpy as np
dm={"price":np.array([1,2,3]),"count":np.array([10,20,
30])}
dfm = pd.DataFrame(dm, index=['a', 'b', 'c'])
print(dfm.loc['a'])
```

Результат:

price 1

count 10

Name: a, dtype: int32

Операція: вибір рядка за індексом

Приклад 6.20.

```
import pandas as pd
import numpy as np
dm={"price":np.array([1, 2, 3]),"count":np.array([10,20,
```

```
30])}
dfm = pd.DataFrame(dm, index=['a', 'b', 'c'])
print(dfm.iloc[2])
```

Результат:

price 3

count 30

Name: c, dtype: int32

Операція: зріз по рядках

Приклад 6.21.

```
import pandas as pd
import numpy as np
dm={"price":np.array([1, 2, 3]),"count":np.array([10,20,
30])}
dfm = pd.DataFrame(dm, index=['a', 'b', 'c'])
print(dfm[0:2])
```

Результат:

	price	count
a	1	10
b	2	20

Операція: вибір рядків, що відповідають умові

Приклад 6.22.

```
import pandas as pd
import numpy as np
dm = {"price":np.array([1, 2, 3]), "count": np.array([10,
20, 30])}
dfm = pd.DataFrame(dm, index=['a', 'b', 'c'])
print(dfm[dfm['count'] >= 20])
```

Результат:

	price	count
b	2	20
c	3	30

6.3. Доступ до даних у структурах Pandas

6.3.1. Два підходи до отримання доступу до даних в Pandas

При роботі з структурами Series і DataFrame з бібліотеки pandas, як правило, використовують два основних способи отримання значень елементів.

Перший спосіб заснований на використанні міток, в цьому випадку робота ведеться через метод .loc. Якщо ви звертаєтеся до відсутньої мітки, то буде згенеровано помилку KeyError.

Такий підхід дозволяє використовувати:

• мітки у вигляді окремих символів ['a'] або чисел [5], числа використовуються як мітки, якщо при створенні структури не був вказаний список з мітками;

• список міток ['a', 'b', 'c'];

• зріз міток ['a': 'c'];

• масив логічних змінних;

• функцію зворотного виклику з одним аргументом.

Другий спосіб заснований на використанні цілих чисел для доступу до даних, він надається через метод .iloc. При використанні .iloc, якщо відбувається звертання до неіснуючого елементу, то генерується помилка IndexError. Логіка використання .iloc дуже схожа на роботу з .loc.

При такому підході можна використовувати:

• окремі цілі числа для доступу до елементів структури;

• масиви цілих чисел [0, 1, 2];

• зрізи цілих чисел [1:4];

• масиви логічних змінних;

• функцію зворотного виклику з одним аргументом.

Залежно від типу використовуваної структури буде змінюватися форма .loc:

• для Series вона має такий вигляд: s.loc [indexer];

• для DataFrame: df.loc [row_indexer, column_indexer].

6.3.2. Використання різних способів доступу до даних

Створимо об’єкти типів `Series` і `DataFrame`, які в подальшому будуть використані для експериментів.

Для цього спочатку імпортуємо бібліотеку і створимо структуру `Series`.

Приклад 6.23.

```
import pandas as pd
sr=pd.Series([10,20,30,40,50], ['a','b','c','d','e'])
print('Вивід одного елемента за міткою')
print(sr['a'])
print('Вивід усього Series')
print(sr)
```

Результат:

Вивід одного елемента за міткою
10
Вивід усього Series
a 10
b 20
c 30
d 40
e 50
dtype: int64

Створимо структуру `DataFrame` і виведемо її на друк.

Приклад 6.24.

```
import pandas as pd
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
dfd = pd.DataFrame(d, index=['a', 'b', 'c'])
print(dfd)
```

Результат:

	price	count	percent
a	1	10	24
b	2	20	51
c	3	30	71

6.3.3. Доступ до даних структури `Series`

Доступ з використанням міток

При використанні міток для доступу до даних можна застосовувати один з наступних підходів:

• перший, коли записуємо ім'я змінної типу `Series` і в квадратних дужках вказуємо мітку, по якій бажаємо звернутися (Наприклад: `s ['a']`);

• другий, коли після імені змінної пишемо `.loc` і далі вказуємо мітку в квадратних дужках (наприклад: `s.loc ['a']`).

Одержуємо елемент за міткою:

Приклад 6.25.

```
import pandas as pd
sr=pd.Series([10,20,30,40,50], ['a','b','c','d','e'])
print('Вивід одного елемента за міткою')
print(sr['a'])
```

Результат:

Вивід одного елемента за міткою
10

Одержуємо набір елементів за мітками:

Приклад 6.26.

```
import pandas as pd
sr=pd.Series([10,20,30,40,50], ['a','b','c','d','e'])
print(sr[['a','c','e']])
```

Результат:

Вивід елементів a, c, e
a 10
c 30
e 50
dtype: int64

Отримання зрізу за мітками

```
import pandas as pd
sr=pd.Series([10,20,30,40,50], ['a','b','c','d','e'])
```

```
print('Вивід зрізу за мітками')
print(sr['a':'d'])
```

Результат:
Вивід зрізу за мітками
a 10
b 20
c 30
d 40
dtype: int64

Доступ з використанням цілочислових індексів

При роботі з цілочисельними індексами індекс можна ставити відразу після імені змінної в квадратних дужках (наприклад: `s[1]`), або можна скористатися `.iloc` (наприклад: `s.iloc [1]`).

Одержуємо елемент за окремим індексом

Приклад 6.27.

```
import pandas as pd
si = pd.Series([10, 20, 30, 40, 50])
print('Вивід елемента за індексом')
print(si[2])
```

Результат:
Вивід елемента за індексом
30

Доступ за списком індексів

Приклад 6.28.

```
import pandas as pd
si = pd.Series([10, 20, 30, 40, 50,60,70])
print('Вивід елемента за списком індексів')
print(si[[1,3,5]])
```

Результат:
Вивід елемента за списком індексів
1 20
3 40
5 60
dtype: int64

Доступ за зрізом індексів

Приклад 6.29.

```
import pandas as pd
si = pd.Series([10, 20, 30, 40, 50,60,70])
print('Вивід елемента за зрізом індексів')
print(si[1:5])
```

Результат:

Вивід елемента за зрізом індексів
1 20
2 30
3 40
4 50
dtype: int64

Доступ з використанням `callable` *функції*

Приклад 6.30.

```
import pandas as pd
si = pd.Series([10, 20, 30, 40, 50,60,70])
a=si[lambda x: x>= 30]
print(a)
```

Результат:

```
2    30
3    40
4    50
5    60
6    70
dtype: int64
```

Доступ з логічним виразом

Приклад 6.31.

```
import pandas as pd
si = pd.Series([10, 20, 30, 40, 50,60,70])
a=si[si>3]
```

Результат:

```
3    40
4    50
5    60
6    70
dtype: int64
```

6.3.4. Доступ до даних структури `DataFrame`

Розглянемо різні варіанти використання міток, які можуть бути як іменами стовпців таблиці, так і іменами рядків.

Отримати доступ до стовпчика

Приклад 6.32.

```
import pandas as pd
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
df = pd.DataFrame(d, index=['a', 'b', 'c'])
print(df['count'])
```

Результат:

```
a    10
b    20
c    30
Name: count, dtype: int64
```

Отримати доступ до списку стовпчиків

Приклад 6.33.

```
import pandas as pd
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
df = pd.DataFrame(d, index=['a', 'b', 'c'])
print(df[['count','percent']])
```

Результат:

```
    count  percent
a      10       24
b      20       51
c      30       71
```

Отримати доступ по зрізу міток

Приклад 6.34.

```
import pandas as pd
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
```

```
df = pd.DataFrame(d, index=['a', 'b', 'c'])
print(df['b':'c'])
```

Результат:

```
   price  count  percent
b      2     20       51
c      3     30       71
```

Доступ через `callable` *функцію*

Підхід в роботі з `callable` функцією для `DataFrame` аналогічний тому, що використовується для `Series`, тільки при формуванні умов необхідно додатково вказувати ім’я стовпця.

Отримання всіх елементів, у яких значення в стовпці `'count'` більше 15:

Приклад 6.35.

```
import pandas as pd
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
df = pd.DataFrame(d, index=['a', 'b', 'c'])
a=df[lambda x: x['count'] > 15]
print(a)
```

Результат:

```
   price  count  percent
b      2     20       51
c      3     30       71
```

Доступ через логічний вираз

Приклад 6.36.

```
import pandas as pd
d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent":
[24, 51, 71]}
df = pd.DataFrame(d, index=['a', 'b', 'c'])
a=df[df['percent']>25]
print(a)
```

Результат:

```
   price  count  percent
b      2     20       51
c      3     30       71
```

6.3.5. Використання атрибутів для доступу до даних в Pandas

Для доступу до даних можна використовувати атрибути структур, в якості яких виступають мітки.

Розглянемо структури Series. Скористаємося вже знайомою нам структурою. Для доступу до елементу через атрибут необхідно вказати його через точку після імені змінної.

Приклад 6.37.

```
import pandas as pd
s=pd.Series([10,20,30,40,50], ['a','b','c','d','e'])
print('вивід a')
print(s.a)
print('вивід d')
print(s.d)
```

Результат:

вивід a
10
вивід d
40

Оскільки структура s має мітки 'a', 'b', 'c', 'd', 'e', то для доступу до елементу з міткою 'a' ми можемо використовувати синтаксис s.a.

Цей же підхід можна застосувати для змінної типу DataFrame.

Отримаємо доступ до колонки 'percent'.

Приклад 6.38.

```
import pandas as pd
d={"price":[1,2,3],"count":[10,20,30],"percent":[24,51,71]}
df = pd.DataFrame(d, index=['a', 'b', 'c'])
print(df.percent)
```

Результат:

a 24

b 51

c 71

Name: percent, dtype: int64

6.4. Робота з пропусками в даних

Бібліотека `pandas` надає можливість отримати випадковий набір даних з уже існуючої структури. Такий функціонал надає як `Series`, так і `DataFrame`. У даних структур є метод `sample ()`, що надає випадкову підвибірку.

Розглянемо застосування випадкової вибірки до структури `Series`.

Приклад 6.39.

```
import pandas as pd
srand=pd.Series([10,20,30,40,50],['a','b','c','d','e'])
print(srand.sample ())
```

Результат:

d 40

dtype: int64

Можна зробити вибірку з декількох елементів, для цього потрібно передати потрібну кількість через параметр `n`.

Приклад 6.40.

```
import pandas as pd
srand=pd.Series([10,20,30,40,50],['a','b','c','d','e'])
print(srand.sample (n=4))
```

Результат:

```
e    50
b    20
a    10
c    30
dtype: int64
```

Також є можливість вказати **частку** від загального числа об’єктів в структурі, використовуючи параметр `frac`.

Приклад 6.41.

```
import pandas as pd
srand=pd.Series([10,20,30,40,50],['a','b','c','d','e'])
print(srand.sample (frac=0.5))
```

Результат:

```
c   30
e   50
dtype: int64
```

Кількість виведених елементів округлюється до меншого цілого.

Існує можливість передати вектор ваг, довжина якого має дорівнювати кількості елементів структури. Сума ваг має дорівнювати одиниці, вага, в даному випадку, це ймовірність появи елемента у вибірці.

У прикладі сформуємо вектор ваг і зробимо вибірку з **чотирьох** елементів.

Приклад 6.42.

```
import pandas as pd
srand = pd.Series([101, 210, 301, 410, 501, 621, 754,
802,991], ['a', 'b', 'c', 'd', 'e','f','g','h','k'])
r=[0.04,0.06,0.1,0.15,0.25,0.1,0.08,0.02,0.2]
print(srand.sample (n=4,weights=r))
```

Результат:

```
d   410
e   501
f   621
c   301
dtype: int64
```

Вибірку можна застосувати і до структури `DataFrame`.

Приклад 6.43.

```
import pandas as pd
dfr = { "price": [1,2,3,5,6],"count":[10,20,30,40,50],
"percent": [24, 51, 71 , 25, 42]}
dfr = pd.DataFrame (dfr)
print(dfr.sample ())
```

Результат:

```
  price  count  percent
0     1    10       24
```

Можна зробити вибірку з декількох елементів через параметр `n`.

```
import pandas as pd
dfr = { "price": [1, 2, 3, 5, 6], "count": [10, 20, 30, 40,
50], "percent": [24, 51, 71 , 25, 42]}
dfr = pd.DataFrame (dfr)
print(dfr.sample (2))
```

Результат:

	price	count	percent
4	6	50	42
3	5	40	25

При роботі з `DataFrame` можна вказати ***вісь*** (`exis`).

Вісь – вісь для вибірки.

0 – вібірка по рядках.

1 – вибірка по стовпцях.

Приклад 6.44.

```
import pandas as pd
dfr = { "price": [1, 2, 3, 5, 6], "count": [10, 20, 30, 40,
50], "percent": [24, 51, 71 , 25, 42]}
dfr = pd.DataFrame (dfr)
print(dfr.sample (n=2,axis=1))
```

Результат:

	percent	price
0	24	1
1	51	2
2	71	3
3	25	5
4	42	6

6.4.1. Додавання елементів у структури

При роботі зі структурами `pandas` часто виникає необхідність додавати нові елементи у структуру. Розглянемо процес модифікації структур на практиці.

Додавання нового елемента в структуру Series.

Приклад 6.45.

```
import pandas as pd
smod=pd.Series([10,20,30,40,50],['a','b','c','d','e'])
print('початкова структура Series')
print(smod)
smod['f']=75
print('модифікована структура Series')
print(smod)
```

Результат:

початкова структура Series

```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

модифікована структура Series

```
a    10
b    20
c    30
d    40
e    50
f    75
dtype: int64
```

Додавання нового елемента в структуру DataFrame.

Приклад 6.46.

```
import pandas as pd
dfmod = { "price": [1, 2, 3, 5, 6], "count": [10, 20, 30,
40, 50], "percent": [24, 51, 71 , 25, 42]}
dfr = pd.DataFrame (dfr)
print('початкова структура DataFrame')
```

```
print(dfmod)
dfmod['value']=2,4,8,16,32
print('модифікована структура DataFrame')
print(dfmod)
```

Результат:

початкова структура DataFrame

	price	count	percent
0	1	10	24
1	2	20	51
2	3	30	71
3	5	40	25
4	6	50	42

модифікована структура DataFrame

	price	count	percent	value
0	1	10	24	2
1	2	20	51	4
2	3	30	71	8
3	5	40	25	16
4	6	50	42	32

6.4.2. Індексація з використанням логічних виразів

В роботі з даними часто необхідно отримувати певну підвибірку з існуючого набору даних. Наприклад, отримати всі товари, знижка на які більше п’яти відсотків, або вибрати з бази інформацію про співробітників чоловічої статі віком більше 40 років.

Цей процес нагадує фільтрацію при роботі з таблицями або отримання вибірки з бази даних. Таку операцію реалізовано в `pandas` і це питання частково розглядалося при індексації. Умовний вираз записується замість індексу в квадратних дужках при зверненні до елементів структури.

Розглянемо застосування функціоналу для структури **`Series`**.

Приклад 6.47.

```
import pandas as pd
slog = pd.Series ([10, 20, 30, 40, 50, 10, 10], [ 'a', 'b', 'c', 'd', 'e', 'f ',' g ']) 
print('Варіант1')
print(slog[slog>=40])
print('Варіант2')
print(slog[slog==10])
print('Варіант3')
print(slog[slog!=50])
print('Варіант4')
print(slog[(slog>=30)&(slog<50)])
```

Результат:

Варіант1
d 40
e 50
dtype: int64
Варіант2
a 10
f 10
g 10
dtype: int64
Варіант3
a 10
b 20
c 30
d 40
f 10
g 10
dtype: int64
Варіант4
c 30
d 40
dtype: int64

При роботі з **DataFrame** вказуємо стовпець, за яким буде проводитися фільтрація (вибірка).

Приклад 6.48.

```
import pandas as pd
dl = { "price": [1, 2, 3, 5, 6], "count": [10, 20, 30, 40,
50], "percent": [24, 51, 71 , 25, 42], "cat": [ "A", "B",
"A", "A", "C"]}
dfl = pd.DataFrame(dl)
print('Повний вивід')
print(dfl)
print('Варіант1')
print(dfl[dfl['price']>3])
print('Варіант2')
print(dfl[dfl['cat']!='A'])
print('Варіант3')
print(dfl[(dfl['percent']>30)&(dfl['count']<50)])
```

Результат:

Повний вивід

	price	count	percent	cat
0	1	10	24	A
1	2	20	51	B
2	3	30	71	A
3	5	40	25	A
4	6	50	42	C

Варіант1

	price	count	percent	cat
3	5	40	25	A
4	6	50	42	C

Варіант2

	price	count	percent	cat
1	2	20	51	B
4	6	50	42	C

Варіант3

	price	count	percent	cat
1	2	20	51	B
2	3	30	71	A

Для формування логічного виразу можна використовувати досить складні конструкції з використанням `map, filter,` лямбда-виразів і т. п.

Приклад 6.49.

```
import pandas as pd
dl = { "price": [1, 2, 3, 5, 6], "count": [10, 20, 30, 40,
50], "percent": [24, 51, 71 , 25, 42], "cat": [ "A", "B",
"A", "A", "C"]}
dfl = pd.DataFrame(dl)
fn = dfl [ "cat"].map (lambda x: x == "A")
print(dfl[fn])
```

Результат:

	price	count	percent	cat
0	1	10	24	A
2	3	30	71	A
3	5	40	25	A

6.4.3. Використання `isin` для роботи з даними в `Pandas`

За структурами даних `pandas` можна будувати масиви з даними типу `boolean`, за яким можна перевірити наявність або відсутність того чи іншого елемента. Найпростіше це показати на прикладі.

Приклад 6.50.

```
import pandas as pd
sil = pd.Series ([10, 20, 30, 40, 50, 10, 10], ['a', 'b',
'c', 'd', 'e', 'f ',' g ']) 
print(sil.isin([10,20]))
```

Результат:

```
a       True
b       True
c       False
```

```
d      False
e      False
f      True
g      True
dtype: bool
```

Робота з `DataFrame` *аналогічна роботі з структурою* `Series`.

Приклад 6.51.

```
import pandas as pd
df = pd.DataFrame({"price":[1, 2, 3, 5, 6], "count": [10,
20, 30, 40, 50], "percent": [24, 51, 71, 25, 42]})
print('Вивід DataFrame')
print(df)
print('Вивід isin')
print(df.isin([1, 3, 25, 30, 10]))
```

Результат:

Вивід DataFrame

	price	count	percent
0	1	10	24
1	2	20	51
2	3	30	71
3	5	40	25
4	6	50	42

Вивід isin

	price	count	percent
0	True	True	False
1	False	False	False
2	True	True	False
3	False	False	True
4	False	False	False

6.4.4. Робота з пропусками даних

Дуже часто великі обсяги даних, які готуються для подальшого аналізу, мають пропуски. Для того, щоб можна було використовувати алгоритми машинного навчання, які будують моделі за цими даними, в більшості випадків необхідно ці пропуски заповнити. Розглянемо механізм заповнення пропущених даних.

Попередньо створимо структуру `DataFrame`, яка міститиме пропущені дані, з використанням популярного в `Data Scince` формату даних `*.csv`.

Відповідно до даного формату створимо рядок, у якому дані відділяються комами.

Приклад 6.52.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
df = pd.read_csv(StringIO(data))
print(df)
```

Результат:

```
  price  count  percent
0     1    10      NaN
1     2    20     51.0
2     3    30      NaN
```

У нашому прикладі у об'єктів з індексами 0 і 2 відсутні дані в полі 'percent'. Відсутні дані позначаються як NaN. Додамо до існуючої структури ще один об'єкт (запис), у якому буде відсутнє значення в полі 'count'.

Приклад 6.53.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
df = pd.read_csv(StringIO(data))
df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print(df)
```

Результат:

```
  price  count  percent
0   1.0   10.0      NaN
1   2.0   20.0     51.0
2   3.0   30.0      NaN
3   4.0    NaN     26.3
```

Звернемося до методів з бібліотеки pandas, які дозволяють швидко визначити наявність елементів NaN в структурах. Якщо таблиця невелика, то можна використовувати бібліотечний метод **isnull.**

Приклад 6.54.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
df = pd.read_csv(StringIO(data))
df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print( pd.isnull(df))
```

Результат:

```
  price  count  percent
0 False  False    True
1 False  False   False
2 False  False    True
3 False   True   False
```

Таким чином ми отримуємо таблицю того ж розміру, але на місці реальних даних в ній знаходяться логічні змінні, які приймають значення False, якщо значення поля в об’єкта є, або True, якщо значення в даному полі – це NaN.

Можна також подивитися детальну інформацію про об’єкт, для цього можна скористатися методом **info()**.

Приклад 6.55.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
df = pd.read_csv(StringIO(data))
df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print(df.info)
```

Результат:

```
<class 'pandas.core.frame.DataFrame'>
```

Int64Index: 4 entries, 0 to 3

Data columns (total 3 columns):

price 4 non-null float64

count 3 non-null float64

percent 2 non-null float64

dtypes: float64(3)

memory usage: 128.0 bytes

None

У нашому прикладі видно, що об'єкт `df` має три стовпці (`count`, `percent` і `price`), при цьому в стовпці `price` всі об’єкти значимі – НЕ `NaN`, в стовпці `count` – один `NaN` об’єкт, в полі `percent` – два `NaN` об’єкта. Можна скористатися таким підходом для отримання кількості `NaN` елементів у записах.

Приклад 6.56.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
df = pd.read_csv(StringIO(data))
df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print('Кількість NaN')
print(df.isnull().sum())
```

Результат:

Кількість NaN

price 0

count 1

percent 2

dtype: int64

6.4.5. Заміна відсутніх даних

Відсутні дані об’єктів можна замінити на конкретні числові значення, для цього можна використовувати метод `fillna()`. Для експериментів будемо використовувати структуру `df`, створену раніше.

Приклад 6.57.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
df = pd.read_csv(StringIO(data))
df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print('Початкова структура')
print(df)
print('Кількість NaN')
print(df.isnull().sum())
print('Замінили NaN на 0')
print(df.fillna(0))
```

Результат

Початкова структура

	price	count	percent
0	1.0	10.0	NaN
1	2.0	20.0	51.0
2	3.0	30.0	NaN
3	4.0	NaN	26.3

Кількість NaN

price 0

count 1

percent 2

dtype: int64

Замінили NaN на 0

	price	count	percent
0	1.0	10.0	0.0
1	2.0	20.0	51.0
2	3.0	30.0	0.0
3	4.0	0.0	26.3

Цей метод не змінює поточну структуру, він повертає структуру **DataFrame**, створену на базі існуючої, з заміною NaN значень на ті, що передані в метод як аргумент. Дані можна заповнити середнім значенням по стовпцю.

Приклад 6.58.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
dfs = pd.read_csv(StringIO(data))
dfs.loc[3] = {'price':4, 'count':None, 'percent':26.3}
print('Замінили NaN на середнє')
print(dfs.fillna(dfs.mean()))
```

Результат

Замінили NaN на середнє

	price	count	percent
0	1.0	10.0	38.65
1	2.0	20.0	51.00
2	3.0	30.0	38.65
3	4.0	20.0	26.30

Залежно від завдання використовується той чи інший метод заповнення відсутніх елементів, це може бути нульове значення, математичне сподівання, медіана і т. п. Для заміни NaN елементів на конкретні значення можна використовувати інтерполяцію, яка реалізована в методі **interpolate ()**, алгоритм інтерполяції задається через аргументи методу.

Приклад 6.59.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n9,0,10\n2,20,51\n3,30,'
dfs = pd.read_csv(StringIO(data))
dfs.loc[3] = {'price':4, 'count':None, 'percent':26.3}
```

```
print('Інтерполяція')
print(dfs.fillna(dfs.interpolate (method='linear')))
```

Результат

Інтерполяція

	price	count	percent
0	9.0	0.0	10.00
1	2.0	20.0	51.00
2	3.0	30.0	38.65
3	4.0	30.0	26.30

6.4.6. Видалення об’єктів / стовпців з відсутніми даними

Часто використовуваний підхід при роботі з відсутніми даними – це видалення записів (рядків) або полів (стовпців), в яких зустрічаються пропуски. Для того, щоб видалити всі об’єкти, які містять значення `NaN`, користуються методом **`dropna()`** без аргументів.

Приклад 6.60.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n9,0,10\n2,20,51\n3,30,'
dfs = pd.read_csv(StringIO(data))
print('До видалення')
print(dfs)
print('Після видалення')
print(dfs.dropna())
```

Результат

До видалення

	price	count	percent
0	9	0	10.0
1	2	20	51.0
2	3	30	NaN

Після видалення

	price	count	percent
0	9	0	10.0
1	2	20	51.0

Замість записів (рядків) можна видалити поля (стовпці), для цього потрібно викликати метод **dropna** з аргументом `axis = 1`.

Приклад 6.61.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n9,0,10\n2,20,51\n3,30,'
dfs = pd.read_csv(StringIO(data))
print('До видалення')
print(dfs)
print('Після видалення')
print(dfs.dropna(axis=1))
```

Результат

До видалення

	price	count	percent
0	9	0	10.0
1	2	20	51.0
2	3	30	NaN

Після видалення

	price	count
0	9	0
1	2	20
2	3	30

`Pandas` дозволяє задати поріг на кількість не `NaN` елементів. У наведеному нижче прикладі будуть видалені всі стовпці, в яких кількість не `NaN` елементів менше трьох.

Приклад 6.62.

```
import pandas as pd
from io import StringIO
data = 'price,count,percent\n9,,10\n2,20,51\n3,30,'
dfs = pd.read_csv(StringIO(data))
```

```
print('До видалення')
print(dfs)
print('Після видалення')
print(dfs.dropna(axis=1, thresh=3))
```

Результат

До видалення

	price	count	percent
0	9	NaN	10.0
1	2	20.0	51.0
2	3	30.0	NaN

Після видалення

	price
0	9
1	2
2	3

Контрольні запитання до розділу 6

1. Особливості структури даних `Series`. Створення `Series` зі списку і словника `Python`.
2. Принципи роботи з елементами струкутури даних `Series`.
3. Особливості структури даних `Series`. Способи створення структури `DataFrame`.
4. Принципи роботи з елементами струкутури даних `DataFrame`.
5. Використання атрибутів для доступу до даних в `Pandas`.
6. Опишіть методи роботи з пропущеними даними.
7. Використання `isin` для роботи з даними в `Pandas`.

7. ОГЛЯД МЕТОДІВ ДОБУВАННЯ ДАНИХ

Існує кілька способів побудови прогнозних моделей із наборів даних, і дослідник даних має розуміти концепції цих методів, а також те, як використовувати код для створення подібних моделей та візуалізацій. Ці методи включають:

Регресія – оцінка взаємозв'язків між змінними шляхом оптимізації зменшення похибки.

Класифікація – визначення, до якої категорії належить об'єкт. Прикладом є класифікація електронної пошти як спаму чи законної або перегляд кредитного рейтингу особи та схвалення чи відмова у видачі кредиту.

Кластерний аналіз – пошук природних груп об'єктів даних на основі відомих характеристик цих даних. Приклад можна побачити в маркетингу, де аналіз може виявити групи споживачів з унікальною поведінкою, які можуть бути застосовані при прийнятті бізнес-стратегії.

Аналіз асоціації та кореляції – шукаємо, чи існують унікальні зв'язки між змінними, які не є одразу очевидними. Прикладом може бути відомий випадок пива та памперсів: чоловіки, які купували памперси наприкінці тижня, набагато частіше купували пиво, тому магазини розміщували їх близько один до одного, щоб збільшити продажі.

Аналіз зовнішніх результатів – вивчення викидів для вивчення потенційних причин та причин згаданих відхилень. Прикладом цього є використання аналізу несанкціонованого доступу при виявленні шахрайства та спроба визначити, чи є модель поведінки поза нормою шахрайською, чи ні.

Добування даних для бізнесу часто виконується за допомогою транзакційної та реальної бази даних, що дозволяє легко використовувати засоби аналізу даних для аналізу. Одним з прикладів може бути онлайновий сервер аналітичної обробки або OLAP, який дозволяє користувачам проводити багатовимірний аналіз на сервері даних. OLAP дозволяють бізнесу запитувати та аналізувати дані без необхідності завантажувати статичні файли даних, що корисно в ситуаціях, коли

база даних зростає щодня. Однак для тих, хто хоче навчитися інтелектуальному аналізу даних і займатися самостійно, ноутбук `iPython` буде ідеально підходити для вирішення більшості завдань з інтелектуального аналізу даних.

Розглянемо, як за допомогою `Python` виконувати добування даних за допомогою двох алгоритмів добування даних: регресія та кластеризація.

7.1. Задачі регресійної оцінки (regression estimation)

Походження терміну «регресія»

Термін «регресія» був введений в 1886 році антропологом Френсісом Гальтоном при вивченні статистичних закономірностей спадковості зросту. Повсякденний досвід підказує, що в середньому зріст дорослих дітей тим більше, чим вище їхні батьки. Але Гальтон виявив, що сини дуже високих батьків часто мають не такий високий зріст. Він зібрав вибірку даних по 928 парах батько-син. Кількісно залежність непогано описувалась лінійною функцією `y=2/3x`, де `x` – відхилення зросту батька від середнього, а `y` – відхилення зросту сина від середнього. Гальтон назвав це явище «регресією до пересічності», тобто, до середнього значення в популяції. Термін «регресія», або рух назад натякав також на нестандартний для того часу хід дослідження: спочатку були зібрані дані, потім по них вгадана модель залежності, тоді як традиційно вчиняли навпаки: дані використовувалися лише для перевірки теоретичних моделей. Це був один з перших випадків моделювання, заснованого виключно на даних. Пізніше термін, що виник в конкретній прикладній задачі, закріпився за широким класом методів відновлення залежностей. Величезна кількість регресійних задач виникає у фізичних експериментах, в промисловому виробництві, в економіці.

7.1.1. Набір даних MNIST

Набір даних MNIST – це дуже популярний набір даних машинного навчання, що складається з 60 000 навчальних і 10 000 тестових зображень у відтінках сірого рукописних цифр розміром 28x28. Ми будемо використовувати його як наш приклад набору даних з метою класифікації зображень цифр.

Найважливішим кроком машинного навчання є підготовка даних.

Цей крок може включати:

завантаження, упорядкування, форматування, перемішування, попередню обробку, створення міні-наборів (minibatch).

Пакет **torchvision** спрощує це, реалізуючи багато з них, що дозволяє укласти ці набори даних у форму, придатну для використання лише за кілька рядків коду.

Завантаження MNIST

Приклад 7.1.

```
import torch
from torchvision import datasets, transforms
mnist_train = datasets.MNIST(root="datasets",
train=True, transform=transforms.ToTensor(), download=True)
mnist_test = datasets.MNIST(root="datasets",
train=False, transform=transforms.ToTensor(),
download=True)
print("Розмір навчального набору MNIST: {}"
.format(len(mnist_train)))
print("Розмір тестового набору MNIST {}"
.format(len(mnist_test)))
```

Набір даних **MNIST** завантажується з такими параметрами:

root – це шлях, де зберігаються навчальні/тестові дані.

train – визначає навчальний або тестовий набір даних.

download=True завантажує дані з Інтернету, якщо вони недоступні в `root`.

transform визначають перетворення ознак з формату PI (Pillow Image) в формат `Tensor PyTorch`

Перетворення формату рисунка

За замовчуванням тензор `image` є тривимірним. «1» у першому вимірі вказує, що зображення має лише один канал, тобто зображення створене відтінками сірого. Ми повинні це змінити для того, щоб візуалізувати зображення за допомогою **imshow.**

Приклад 7.2.

```
# Вибираємо  4-й приклад з 0-індексованого навчального набору
image, label = mnist_train[3]
# Відображаємо рисунок
print("Базова форма рисунка: {}".format(image.shape))
image = image.reshape([28,28])
print("Перетворена форма рисунка: {}".format(image.shape))
plt.imshow(image, cmap="gray")
# Друк мітки
print("Мітка цього рисунка: {}".format(label))
```

Підготовка даних до навчання за допомогою DataLoaders

При обробці даних нашого набору ми зчитуємо кожен об'єкт, який представлений набором ознак, і відповідну йому мітку послідовно один за одним.

Під час навчання моделі використовуємо «міні-пакети» ("minibatches").

У кожну епоху навчання необхідно перетасовувати дані для того, щоб зменшити ризик перенавчання моделі.

При обробці даних також важливо використати мультипроцесні можливості Python для прискорення пошуку даних.

Всі згадані функції реалізує **DataLoader**.

DataLoader – це ітератор, який абстрагує ці складні задачі у простому API.

Приклад 7.3.

```
train_loader = torch.utils.data.DataLoader(mnist_train,
batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(mnist_test,
batch_size=100, shuffle=False)
data_train_iter = iter(train_loader)
images, labels = data_train_iter.next()
print("Форма для minibatch зображень:
```

```
{}".format(images.shape))
print("Форма для minibatch міток: {}".format(labels.shape))
```

Щоб отримати прогнозовані ймовірності кожної цифри, спочатку почнемо з ймовірності цифри 1. Для нашої моделі можливо почати із застосування лінійного перетворення. Тобто ми множимо кожен піксель x_i вхідного вектора рядка на вагу $w_{i,j}$, підсумовуємо їх усі разом, а потім додаємо зсув b_j. Це еквівалентно скалярному добутку між ваговими коефіцієнтами класу "1" і вхідними параметрами:

$$y_j = \sum_i x_i w_{i,j} + b_j \text{, де } j = 0,1,2,3,4,5,6,7,8,9$$

У матричній формі: $Y = XW + B$. Отримали 10 класів, оскільки ми маємо 10 цифр.

У нашому конкретному прикладі розмір minibatch *m* дорівнює 100, розмірність даних – 28×28=784, а кількість класів *c* – 10. Хоча X і Y є матрицями через пакетування, вони часто задаються змінними у нижньому регістрі.

7.1.2. Початкова ініціалізація параметрів моделі

Ми будемо використовувати *x* і *y* у коді. Параметри моделі: матриця W і зсув B.

Коли ми говоримо, що хочемо «навчити модель», ми насправді намагаємося знайти хороші значення для кожного елемента в W і B.

Перш ніж почати навчання, нам потрібно ініціалізувати наші параметри до певного значення, як відправної точки.

Тут ми насправді не знаємо, які найкращі значення, тому ініціалізуємо W випадковим чином і встановлюємо B у нульовий вектор.

Приклад 7.4.

```
# Випадкова ініціалізація W
W = torch.randn(784, 10)/np.sqrt(784)
W.requires_grad_()
# Нульова ініціалізація вектора b
b = torch.zeros(10, requires_grad=True)
```

7.1.3. Обчислення y з урахуванням градієнтного спуску для параметрів

Оскільки W і b є параметрами, які ми хочемо вивчити, встановлюємо для **`requires_grad`** значення **`True`.** Це повідомляє автограду PyTorch відстежувати градієнти для цих двох змінних і всіх змінних залежно від W і b . За допомогою цих параметрів моделі ми обчислюємо y:

```
y = torch.matmul(x, W) + b  # Лінійне перетворення W і b
```

Наприклад, ми можемо побачити, як виглядають прогнози для першого прикладу в нашому міні-пакеті. Пам'ятайте: чим більше число, тим більше модель «думає», що вхід x належить до цього класу.

```
print(y[0][:])
```

В результаті отримаємо зріз тензора:

```
tensor([-0.2828,  0.2954,  0.2179,  0.3310, -0.4076,
    -0.0848,  0.3806, -0.3472, 0.4987,  0.1375],
grad_fn=<SliceBackward0>)
```

Застосування `softmax()`

Інтерпретувати ці значення y (або логіти) можливо як ймовірності, якщо нормалізуємо їх до позитивних і додаємо до 1. У логістичній регресії робимо це за допомогою `softmax`:

$$p(y_i) = \text{softmax}(y_i) = \frac{\exp(y_i)}{\sum_j y_j}$$

Оскільки область значень експоненціальної функції завжди невід'ємна, і оскільки ми нормалізуємо сумою, `softmax` досягає бажаної властивості генерувати значення від 0 до 1, що становлять суму до 1. Якщо ми подивимося на випадок з лише двох класів, то можна зробити висновок, що `softmax` є багатокласовим розширенням бінарної сигмоїдної функції.

Два варіанта обчислення softmax

Можливо обчислити softmax самостійно, використовуючи наведену вище формулу, але PyTorch вже має функцію `softmax` в **`torch.nn.functional`**.

Приклад 7.5

```
# Варіант 1: Обчислюємо softmax з рівняння
py_eq = torch.exp(y) / torch.sum(torch.exp(y), dim=1,
keepdim=True)
print("py[0] з рівняння: {}".format(py_eq[0]))
# Варіант 2: Обчислюємо softmax з використанням
torch.nn.functional
import torch.nn.functional as F
py = F.softmax(y, dim=1)
print("py[0] з torch.nn.functional.softmax:
{}".format(py[0]))
```

Результат

```
py[0] з рівняння: tensor([0.0668, 0.1191, 0.1102, 0.1234,
0.0590, 0.0814, 0.1297, 0.0626, 0.1460, 0.1017],
grad_fn=<SelectBackward0>)
py[0] з torch.nn.functional.softmax: tensor([0.0668,
0.1191, 0.1102, 0.1234, 0.0590, 0.0814, 0.1297, 0.0626,
0.1460, 0.1017], grad_fn=<SelectBackward0>)
```

Ми закінчили прямий перехід нашої моделі. Отримали ймовірності, які модель вважає вхідними для кожного з 10 класів.

7.1.4. Крос-ентропійна функція втрат

Згадаємо, що ми ще не налаштували значень параметрів W і b. Адже їх ініціалізували випадковим чином. Перш ніж відкорегувати будь-які ваги, нам потрібен спосіб виміряти, як працює модель. Зокрема, ми збираємося виміряти, наскільки погано працює модель. Робимо це за допомогою функції втрат, яка приймає прогноз моделі і повертає одне число (тобто скаляр), що підсумовує

продуктивність моделі. Це число буде інформувати про те, як ми корегуємо параметри моделі.

Функція втрат, яку ми зазвичай використовуємо при класифікації, – це крос-ентропійна концепція з теорії інформації. Пояснення того, що саме представляє крос-ентропія, дещо виходить за рамки цього курсу, але ви можете розглядати це як спосіб кількісної оцінки того, наскільки один розподіл y' віддалений від іншого y.

$$H_{y'}(y) = -\sum_i y'_i \log(y_i)$$

У нашому випадку y – це набір ймовірностей, передбачених моделлю (*py* вище); y' – цільовий розподіл. Цільовий розподіл – це справжня мітка, яку ми хотіли, щоб модель передбачила.

Перехресна ентропія не тільки фіксує, наскільки правильними (максимальна ймовірність відповідає правильній відповіді) є відповіді моделі, але й враховує, наскільки вони впевнені (висока впевненість у правильних відповідях). Це спонукає модель створювати дуже високі ймовірності правильних відповідей, знижуючи ймовірності неправильних відповідей, замість того, щоб просто задовольнятися тим, що це `argmax`.

Зосереджуємось на навчанні з учителем, у якому ми маємо мітки. Наш `DataLoader` автоматично включає відповідні мітки для кожного з наших входів. Ось кількість міток та їх значення для першого отриманого `minibatch`:

Приклад 7.6

```
print(labels.shape)
print(labels)
```

Результат

```
torch.Size([100])
tensor([
3, 9, 4, 9, 0, 8, 3, 0, 4, 4, 0, 4, 7, 1, 3,
0, 2, 6, 5, 3, 2, 3, 9, 5, 2, 7, 8, 5, 6, 2,
8, 6, 2, 3, 1, 4, 0, 6, 9, 0, 8, 2, 6, 1, 1,
```

```
7, 5, 9, 4, 8, 9, 7, 3, 2, 4, 1, 9, 6, 0, 2,
8, 9, 7, 7, 8, 9, 2, 7, 2, 3, 8, 4, 3, 0, 5,
9, 8, 0, 3, 9, 3, 8, 3, 3, 6, 7, 2, 7, 1, 4,
3, 9, 9, 9, 8, 2, 3, 6, 4, 1])
```

Два варіанта обчислення крос-ентропійної втрати

Як і операція `softmax`, ми можемо реалізувати перехресну ентропію безпосередньо з рівняння, використовуючи вихід `softmax`. Однак, як і у випадку з `softmax`, у `torch.nn.functional` вже реалізована перехресна втрата ентропії.

Приклад 7.7.

```
# Варіант 1: Обчислення крос-ентропійної функції з виразу
cross_entropy_eq = torch.mean(-
torch.log(py_eq)[range(labels.shape[0]),labels])
print("крос-ентропія з виразу:
{}".format(cross_entropy_eq))
# Варіант 2: Обчислення крос-ентропійної функції з
torch.nn.functional
cross_entropy = F.cross_entropy(y, labels)
print("крос-ентропія з torch.nn.functional.cross_entropy:
{}".format(cross_entropy))
```

Результат

крос-ентропія з виразу: 2.3316853046417236

крос-ентропія з torch.nn.functional.cross_entropy: 2.3316855430603027

7.1.5.Зворотний прохід

Тепер, коли ми маємо втрату як спосіб кількісної оцінки того, наскільки погано працює модель, ми можемо покращити нашу модель, змінивши параметри таким чином, щоб мінімізувати втрати. Для нейронних мереж звичайний спосіб зробити це – зворотне поширення: беремо градієнт втрати відносно W і b і робимо крок у напрямку, який зменшує втрати.

Спочатку потрібно створити оптимізатор. Існує багато варіантів, але оскільки логістична регресія досить проста, ми будемо використовувати стандартний стохастичний градієнтний спуск (SGD), який робить наступне оновлення:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L,$$

де θ – параметр, α – швидкість навчання (величина кроку), $\nabla_\theta L$ – градієнт втрат по відношенню до θ.

При створенні параметрів W і b ми вказували, що вони вимагають градієнтів. Щоб обчислити градієнти для W і b, викликаємо функцію **`backward()`** для крос-ентропійної функції втрат.

Приклад 7.8.

```
optimizer = torch.optim.SGD([W,b], lr=0.1) # Оптимізатор
cross_entropy.backward()
```

Кожна зі змінних, для яких потрібні градієнти, тепер накопичила градієнти. Ми бачимо це, наприклад, на b. Початкові нульові зсуви змінилися.

```
b.grad
```

Результат

```
tensor([ 0.0098,  0.0036, -0.0097,  0.0718, -0.0258, -
0.0697,  0.0434,  0.0383, 0.0088, -0.0703])
```

Для оновлення значень параметрів з урахуванням градієнтів виконаємо крок оптимізації. Ми встановили швидкість навчання на 0,1, тому b було оновлено на `-0.1 x b.grad`:

Приклад 7.9. Крок оновлення параметрів

```
optimizer.step()
b
```

Результат

```
tensor([-0.0028, -0.0004,  0.0026, -0.0182,  0.0060,
0.0190,
-0.0120, -0.0092,-0.0028,  0.0178], requires_grad=True)
```

Тепер ми успішно навчалися на одному `minibatch`. Однак використання одного `minibatch` недостатньо для досягнення бажаної точності. На даний

момент ми навчили модель на 100 прикладах із 60 000 у навчальному наборі. Нам потрібно буде повторити цей процес, щоб отримати більше даних.

Але слід пам'ятати: градієнти, які обчислені за допомогою **backward()**, не замінюють старі значення; натомість вони накопичуються. Тому вам потрібно очистити буфери градієнта перед обчисленням градієнтів для наступного minibatch.

Приклад 7.10.

```
print("b.grad перед zero_grad(): {}".format(b.grad))
optimizer.zero_grad()
print("b.grad після zero_grad(): {}".format(b.grad))
```

Результат

```
b.grad before zero_grad():
tensor([ 0.0325, -0.0525, -0.0653, -0.0147,  0.0138,
0.0380, -0.0032,  0.0205,0.0342, -0.0032])
b.grad after zero_grad():
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

7.1.6. Тренування моделі

Щоб навчити модель, нам просто потрібно повторити те, що ми щойно зробили для додаткових пакетів minibatch із навчального набору.

Як підсумок, необхідно виконати такі кроки:

-Отримайте пакет minibatch

-Обнуліть градієнти в буферах для W і b

-Виконайте прямий перехід (розрахувати прогноз, обчислити втрати)

-Виконайте зворотний прохід (обчислюйте градієнти, виконайте крок SGD)

Один раз проходження всього набору даних називають епохою. У багатьох випадках ми тренуємо нейронні мережі для кількох епох.

Ми також обгортаємо **train_loader** з **tqdm**. Це не обов'язково, але це додає зручний індикатор прогресу задля забезпечення відстеження прогресу навчання.

Наведена реалізація тільки однієї епохи.

Приклад 7.11.

```
# Ітерування по пакетах minibatch з навчального набору
for images, labels in tqdm(train_loader):
    # Обнулення градієнта
    optimizer.zero_grad()
    # Прямий прохід
    x = images.view(-1, 28 * 28)
    y = torch.matmul(x, W) + b
    cross_entropy = F.cross_entropy(y, labels)
    # Зворотний прохід
    cross_entropy.backward()
    optimizer.step()
```

Результат

Перевірка

Для кожного зображення в нашому тестовому наборі ми запускаємо дані через модель і приймаємо цифру, до якої маємо найвищу впевненість, як нашу відповідь.

Потім ми обчислюємо точність, дивлячись, наскільки ми отримали правильні дані.

Ми збираємося обернути оцінку за допомогою **`torch.no_grad()`**, оскільки нас не цікавить обчислення градієнтів під час оцінювання. Вимкнувши двигун **`autograd`**, ми можемо прискорити оцінку.

Приклад 7.12.

```
correct = 0
total = len(mnist_test)
with torch.no_grad():
    # Ітерування по пакетах minibatch з навчального набору
    for images, labels in tqdm(test_loader):
```

```
        # Прямий прохід
        x = images.view(-1, 28 * 28)
        y = torch.matmul(x, W) + b
        predictions = torch.argmax(y, dim=1)
        correct += torch.sum((predictions ==
labels).float())
print('Точність: {}'.format(correct / total))
```

Як бачимо, модель вивчила шаблон для кожної цифри.

Пам’ятайте, що наша модель використовує скалярний добуток між вагами кожної цифри та введених даних.

Отже, чим більше введені дані збігаються з шаблоном для цифри, тим вищим буде значення скалярного добутку для цієї цифри, що збільшує ймовірність того, що модель передбачає цю цифру.

Повний код

Вся модель з повним визначенням, навчанням та оцінкою моделі (але за вирахуванням візуалізації ваг) як код, що виконується незалежно один від одного.

Примітка. Точність з повної версії безпосередньо вище від покрокової версії, тому може давати дещо іншу точність тесту. Ми навчали нашу модель зі стохастичним градієнтним спуском (SGD), при цьому слово «стохастичний» підкреслює, що навчання за своєю суттю є випадковим процесом.

Приклад 7.13.

```
import numpy as np
import torch
import torch.nn.functional as F
from torchvision import datasets, transforms
from tqdm.notebook import tqdm
# Завантаження даних
mnist_train = datasets.MNIST(root="./datasets", train=True,
transform=transforms.ToTensor(), download=True)
mnist_test = datasets.MNIST(root="./datasets", train=False,
```

```
transform=transforms.ToTensor(), download=True)
train_loader = torch.utils.data.DataLoader(mnist_train,
batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(mnist_test,
batch_size=100, shuffle=False)
## Навчання
# Ініціалізація параметрів
W = torch.randn(784, 10) / np.sqrt(784)
W.requires_grad_()
b = torch.zeros(10, requires_grad=True)
# Оптимізатор
optimizer = torch.optim.SGD([W, b], lr=0.1)
# Ітерування по пакетах minibatch з навчального набору
for images, labels in tqdm(train_loader):
    # Обнулення градієнта
    optimizer.zero_grad()
    # Прямий прохід
    x = images.view(-1, 28 * 28)
    y = torch.matmul(x, W) + b
    cross_entropy = F.cross_entropy(y, labels)
    # Зворотний прохід
    cross_entropy.backward()
    optimizer.step()
## Перевірка
correct = 0
total = len(mnist_test)
with torch.no_grad():
    # Iterate through test set minibatchs
    for images, labels in tqdm(test_loader):
        # Forward pass
```

```
        x = images.view(-1, 28 * 28)
        y = torch.matmul(x, W) + b
        predictions = torch.argmax(y, dim=1)
        correct += torch.sum((predictions ==
labels).float())
print('Перевірка точності: {}'.format(correct / total))
```

7.2. Задача кластеризації

Гіпотеза компактності

Задача класифікації:

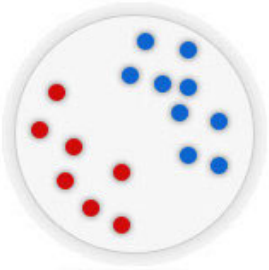
X – об'єкти, Y – відповіді;

$X_m = (x_i, y_i)_{i=1}^m$ – навчальна вибірка;

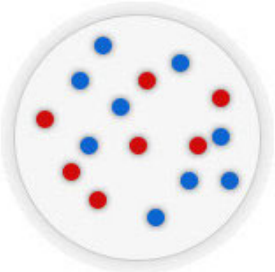
Гіпотеза компактності (для класифікації):

близькі об'єкти, як правило, лежать в одному класі.

Гіпотеза компактності виконується:



Гіпотеза компактності не виконується:



7.2.1. Формалізація поняття відстані

Евклідова метрика $$\rho(x, x_i) = \sqrt{\sum_{j=1}^{n} \left| x^j - x_i^j \right|^2} \qquad (7.1)$$

Узагальнена метрика Мінковського: $$\rho(x,x_i)=\sqrt[p]{w_j\sum_{j=1}^{n}\left|x^j-x_i^j\right|^p} \qquad (7.2)$$

$x=\left(x^1,x^2,...,x^n\right)$ – вектор ознак об'єкта x ,

$x=\left(x_i^1,x_i^2,...,x_i^n\right)$ – вектор ознак про об'єкт x_i,

$\left(w_1,w_2,...,w_n\right)$ – ваги ознак, які можуть змінюватись при навчанні.

7.2.2. Метод найближчого сусіда і його узагальнення

Дано:

1. X – множина об'єктів;

2. $\rho: X\times X\to[0,\infty)$ – функція відстані.

$$X\times X=\left\{(x_1x_1),(x_1,x_2),...,\left(x_1,x_{|X|}\right),(x_2,x_1),(x_2,x_2),...,\left(x_2,x_{|X|}\right),...,\left(x_{|X|},x_1\right),...\right\}$$

3. Існує цільова залежність

$$y^*: X\to Y, \qquad (7.3)$$

значення якої відомі тільки на об'єктах навчальної вибірки:

$$X_m=\left(x_i,y_i\right)_{i=1}^{m},\ y_i=y^*\left(x_i\right). \qquad (7.4)$$

4. Множина класів Y скінченна.

Задача.

Потрібно побудувати такий алгоритм класифікації

$$a: X\to Y, \qquad (7.5)$$

що апроксимує цільову залежність $y^*\left(x\right)$ на всій множині X.

7.2.3. Узагальнений метричний класифікатор

Розглянемо довільний об'єкт $u\in X$

Розташуємо елементи навчальної вибірки $x_1,x_2,...,x_m$ за зростанням відстаней до u та перенумеруємо їх:

$$\rho\left(u,x_u^{(1)}\right)\le\rho\left(u,x_u^{(2)}\right)\le...\le\rho\left(u,x_u^{(m)}\right), \qquad (7.6)$$

де $x_u^{(i)}$ – це i – й сусід об’єкта u.

Відповідь на $i-$му сусіді об’єкта u визначимо:

$$y_u^{(i)} = y^*\left(x_u^{(i)}\right). \tag{7.7}$$

Таким чином, будь-який об'єкт $u \in X$ породжує свою перенумерацію вибірки.

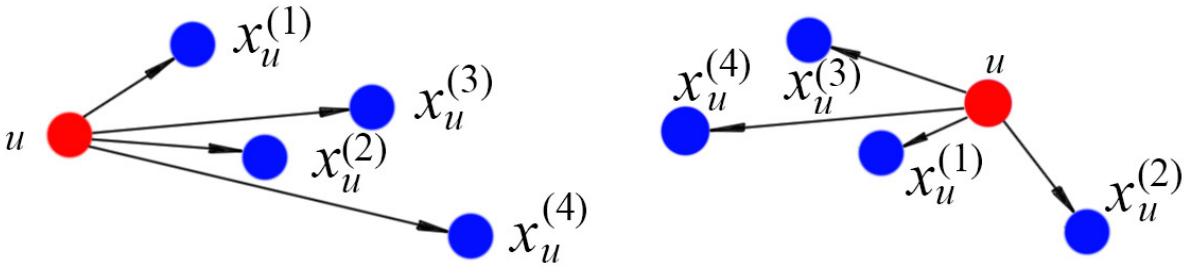


Рис. 7.2. Перенумерація вибірки

7.2.4. Метричний алгоритм класифікації

Визначення. Метричний алгоритм класифікації з навчальною вибіркою X_m відносить об'єкт u до того класу $y \in Y$, для якого сумарна вага найближчих навчальних об’єктів $\Gamma_y\left(u, X_m\right)$ максимальна:

$$\begin{gathered} a\left(u; X_m\right) = \arg\max_{y \in Y} \Gamma_y\left(u, X_m\right); \\ \Gamma_y\left(u, X_m\right) = \sum_{i=1}^{m}\left[y_u^{(i)} = y\right] w(i,u) \end{gathered}, \tag{7.8}$$

де вагова функція $w(i,u)$ оцінює ступінь важливості i-го сусіда для класифікації об'єкта u.

Зазначимо, що для ближчого сусіда $w(i,u)$ може бути більшим.

Функцію $\Gamma_y\left(u, X_m\right)$ називають оцінкою близькості об'єкта u до класу y.

Алгоритм $a\left(u; X_m\right)$ будує локальну апроксимацію вибірки X_m, причому обчислення відкладаються до моменту, поки не стане відомий об'єкт u.

Із цієї причини метричні алгоритми відносять до методів ледачого навчання (*lazy learning*).

Метричні алгоритми класифікації відносять також до методів навчання по прецедентах (*case-based reasoning, CBR*).

На запитання «чому об'єкт *i* був віднесений до класу y ?» алгоритм може дати зрозуміле експертам пояснення: «тому, що є схожі з ним прецеденти класу y», і пред'явити список цих прецедентів.

Вибираючи вагову функцію $w(i,u)$, можна одержувати різні метричні класифікатори, які докладно розглядаються нижче.

7.2.5. Метод найближчих сусідів

Алгоритм найближчого сусіда (*nearest neighbor, NN*) відносить об'єкт, що підлягає класифікації, $u \in X_m$ до того класу, до якого належить найближчий навчальний об'єкт:

$$\{w(1,u)\}; \quad a(u;X_m) = y_u^{(1)} \tag{7.9}$$

Це найпростіший класифікатор.

Навчання NN зводиться до запам'ятовування вибірки X_m. Єдина перевага цього алгоритму – простота реалізації. Недоліків набагато більше:

1. Нестійкість до погрішностей. Якщо серед навчальних об'єктів є *викид* – об'єкт, що перебуває в оточенні об'єктів чужого класу, то не тільки він сам буде класифікований невірно, але й ті навколишні його об'єкти, для яких він виявиться найближчим.

2. Відсутність параметрів, які можна було б налаштувати по вибірці. Алгоритм повністю залежить від того, наскільки вдало обрана метрика ρ.

3. У результаті – низька якість класифікації.

7.2.6. Метод k найближчих сусідів

Алгоритм k найближчих сусідів (*k nearest neighbors, kNN*).

Мета: згладити вплив викидів.

Віднесемо об'єкт u до того класу, елементів якого виявиться більше серед найближчих k сусідів $x_u^{(i)}$, $i = 1,2,...k$:

$$\{w(i,u) | i \le k\}; \quad a(u;X_m,k) = \arg\max_{y \in Y} \sum_{i=1}^{k} \left[y_u^{(i)} = y \right] \tag{7.10}$$

При $k=1$ цей алгоритм збігається з попереднім, отже, нестійкий до шуму.

При $k=m$, навпаки, він надмірно стійкий і вироджується в константу.

Таким чином, крайні значення k є небажаними.

Як знайти оптимальне значення k?

Вибір оптимального k.

На практиці оптимальне значення параметра k визначають за критерієм ковзного контролю *з виключенням об'єктів по одному* (*leave-one-out volidation, LOOV*).

Для кожного об'єкта $x_i \in X_m$ перевіряється, чи правильно він класифікується по своїх k найближчих сусідах.

$$LOOV\left(k, X_m\right) = \sum_{i=1}^{m} \left[a\left(x_i; X_m \setminus \{x_i\}, k\right) \neq y_i \right] \to \min_k \qquad (7.11)$$

Якщо об'єкт x_i, що підлягає класифікації, не виключати з навчальної вибірки, то найближчим сусідом x_i завжди буде сам x_i, і мінімальне (нульове) значення функціонала $LOOV\left(k\right)$ буде досягатися при $k=1$.

Існує й альтернативний варіант методу kNN : у кожному класі вибирається k найближчих до u об'єктів, і об'єкт u відносять до того класу, для якого середня відстань до k найближчих сусідів є мінімальною.

Задача кластеризації на основі k – середніх працює наступним чином:

1. Починаємо із випадково вибраного набору k центроїдів (передбачувані центри для k кластерів).

2. Визначаємо, яке спостереження знаходиться в якому скупченні, виходячи з того, до якого центроїда воно найближче (використовуючи квадрат евклідової відстані).

3. Перераховуємо центроїди кожного скупчення, мінімізуючи квадратну евклідову відстань до кожного спостереження в скупченні.

4. Повторюємо 2. і 3., поки члени скупчень (а отже, і положення центроїдів) більше не змінюватимуться.

Контрольні запитання до розділу 7

1. Які методи групи методів для побудови прогнозних моделей із наборів даних вам відомі?

2. Які основні пакети необхідно завантажити для вирішення задачі регресії?

3. Які способи обчислення градієнтного спуску вам відомі?

4. Поясніть необхідність застосування функції втрат при навчанні моделі.

5. Які варіанти обчислення функції втрат ви знаєте?

6. Перерахуйте кроки, які необхідно виконати для того, щоб навчити модель.

7. В чому полягає ідея гіпотези компактності?

8. Опишіть суть методу найближчих сусідів.

9. Дайте визначення матричного алгоритму кластеризації.

10. Опишіть альтернативний варіант методу kNN, який розглянутий у цьому підручнику.