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

М. А. Новотарський

ОСНОВИ ПРОГРАМУВАННЯ АЛГОРИТМІЧНОЮ МОВОЮ PYTHON

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за освітньою програмою
«Комп'ютерні системи та мережі»,
за спеціальністю 123 «Комп'ютерна інженерія»*

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент: *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.) за поданням Вченої ради Факультету інформатики та обчислювальної техніки (протокол №11 від 11.07. 2022 р.)

Електронне мережеве навчальне видання

ОСНОВИ ПРОГРАМУВАННЯ АЛГОРИТМІЧНОЮ МОВОЮ PYTHON

Основи програмування алгоритмічною мовою Python [Електронний ресурс]: навч. посіб. для студ. освітньої програми «Комп'ютерні системи та мережі» спеціальності 123 «Комп'ютерна інженерія» / М.А. Новотарський – Електронні текстові дані (1 файл: 18 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 701с.

Навчальний посібник створений за матеріалами лекцій з курсу «Програмування-1». Він містить базові відомості про технічне та програмне забезпечення сучасних комп'ютерів. Розглянуто класифікацію та рівні програмного забезпечення. Основна увага приділена основам програмування алгоритмічною мовою Python. Знайомство з Python починається з огляду основних характеристик, історії виникнення цієї мови програмування. Розглянуто типи даних і правила іменування змінних, способи їх типізації. Описано способи використання різних типів операторів, та особливості їх роботи з різними типами змінних. Детально розглянуто стандартні функції та методи, які дозволяють створення, модифікацію та видалення змінних кожного з відомих типів. Окремо приділено увагу функціям та методам, що використовуються при роботі з файлами та модулями. Завершується курс знайомством з основними характеристиками графічної бібліотеки tkinter, яка дозволяє створювати застосунки з графічним інтерфейсом.

Посібник може бути корисним для інженерів та студентів технічних спеціальностей.

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

ЗМІСТ

1. БАЗОВІ ВІДОМОСТІ ПРО ТЕХНІЧНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

1.1. Технічне забезпечення комп’ютерів

1.1.1. Вступ

Стати програмістом – це означає навчитися плавати в безкрайнім морі інформації, пов'язаної з технологіями програмування, фреймворками, середовищами програмування, версіями мови і т. д.

Навчитися плавати – це означає навчитися тим підходам, які є загальними для даної професії. На щастя для нас, практично всі мови програмування засновані на однакових принципах, мають подібні представлення структур даних і подібні прийоми їх обробки.

Чому так відбулося? На це питання є дуже проста відповідь – тому, що всі мови програмування створені для розмови з одним і тим самим об'єктом – комп'ютером.

Комп’ю́тер (англ.*computer* — «обчислювач») — обладнання або система, здатна виконувати задану чітко визначену змінювану послідовність операцій. Це найчастіше операції чисельних розрахунків і маніпулювання даними. В останні роки величезної популярності набрало добування даних.

Названі операції виконує обладнання комп’ютера (hardware): обладнання вводу, системний блок, обладнання виводу.

Обладнання вводу призначене для введення інформації у пам’ять комп’ютера і може включати велику кількість різних пристроїв. Наприклад, клавіатура, мишка, сканер, ведеокамера та інші.

Системний блок комп’ютера виконує обробку даних, тобто перетворює введену інформацію у вихідні дані шляхом виконання алгоритмів.

Вихідні дані стають доступні користовичу завдяки наявності обладнання виводу, яке може включати монітор, принтер або мультимедійні пристрої.

1.1.2. Побутові комп'ютери

Побутові комп'ютери представлені такими виробами		
		
Персональний комп'ютер (*мінімальний набір обладнання: системний блок, монітор, мишка й клавіатура*)	Ноутбук (laptop) (*усе обладнання інтегроване в одному виробі*)	Планшет (tablet) (*обладнання вводу сполучене з обладнанням виводу*)

Рис 1.1. Приклади побутових комп’ютерів

Основний вузол персонального комп'ютера – системний блок. Приблизний його склад показаний на малюнку.

Рис 1.2 Вміст системного блоку

Блок живлення (БП). Це обладнання, що перетворює мережеву напругу змінного струму 220 вольт у ряд низьких напруг постійного струму. За його охолодження відповідає вентилятор.

Блок живлення призначений для живлення материнської плати, плат розширення, вінчестера, CD/ DVD-дисководу, тобто всього обладнання системного блоку.

Рис 1.3. Блок живлення

Оптичний привід — обладнання для запису й відтворення CD/DVD-дисків. До нього надходять електричні шнури від блоку живлення й шина даних від материнської плати. DVD-ROM (RW) служить для читання й запису лазерних CD/DVD-дисків. Компакт-диски можуть містити комп'ютерні програми й ігри, музичні твори, відеофільми і т. д.

Рис. 1.4. Оптичний привід

Процесор. Це велика мікросхема, на яку кріпиться кулер (металевий радіатор з вентилятором). Процесор — головна мікросхема комп'ютера, призначена для виконання різних арифметичних і логічних операцій (інструкцій і програм). Установлюється на материнську плату за допомогою спеціального конектора.

Рис. 1.5. Центральний процесор

Материнська плата. Системна (материнська) плата це друкована плата, у яку встановлюються основні компоненти комп'ютера.

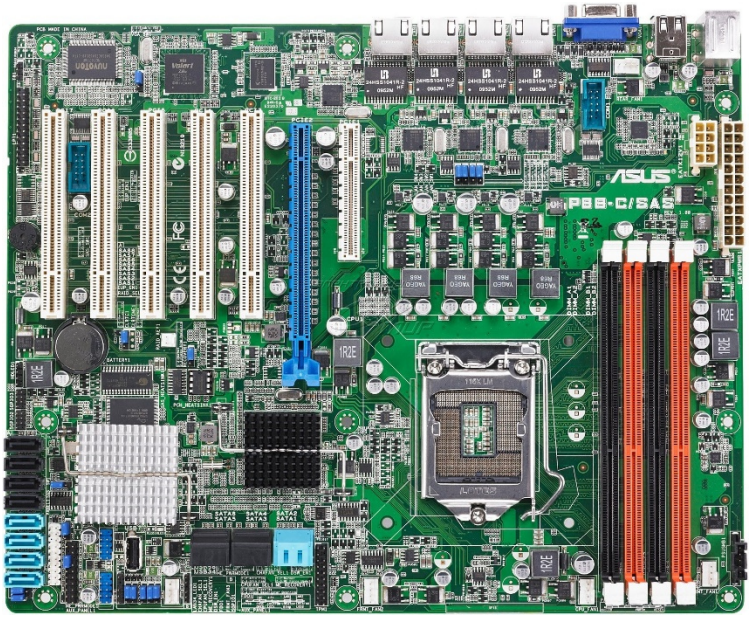


Рис. 1.6. Материнська плата

Відеокарта. Відеокарта призначена для перетворення даних, вироблюваних комп’ютером, у сигнал, відображуваний монітором. HDMI (High-Definition Multimedia Interface)

Рис. 1.7. Відеокарта

Жорсткий диск (вінчестер). Пристрій для зберігання основної інформації на ПК. Вінчестер (накопичувач на жорстких дисках) є місцем зберігання операційної системи, програм, додатків, баз даних, ігор і т. д.
На відміну від оперативної пам’яті, при вимиканні живлення інформація, записана на вінчестер, не губиться. **SATA** (англ. Serial *Advanced Technology Attachment*) — послідовний інтерфейс обміну даними з накопичувачами інформації (як правило, з жорсткими дисками). **SATA** є розвитком інтерфейсу ATA (IDE), який після появи **SATA** був перейменований в PATA (Parallel ATA).

Рис. 1.8. Жорсткий диск

Оперативна пам'ять . Це мікросхема, у якій тимчасово зберігаються дані й команди, необхідні процесору для виконання ним різних операцій. Пам'ять установлюється у спеціальні слоти на материнській платі.
SDRAM (англ. ***S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory* — синхронна динамічна пам'ять з довільним доступом) — тип запам'ятовуючого

пристрою, що використовується в комп'ютерній техніці. Найпоширенішим стандартом пам'яті SDRAM є DDR.

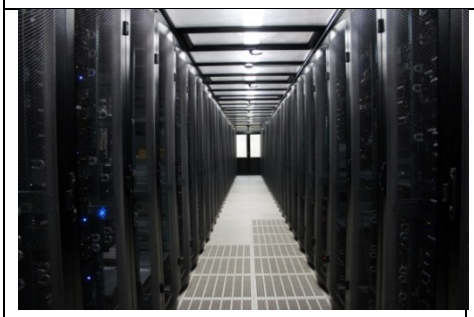
Рис. 1.9. Оперативна пам’ять

Кулер процесора. Кулер установлюється в задній (іноді у верхній) частині системного блоку на гвинти або спеціальні гумові кріплення. Його шнур живлення підключається до материнської плати.

Рис. 1.10. Кулер процесора

1.1.3. Професійні комп'ютери

Центри обробки даних призначені для підтримки сервісів Інтернет або для розв'язування задач колосальної обчислювальної складності.

Дата-центри		
		
Гермозона ЦОД	Операторська кімната	Охолодження

(центр обробки даних)		

Системний блок професійних комп'ютерів конструктивно відрізняється від побутових комп'ютерів. Важливу роль тут відіграють геометричні розміри.

		
Сервер	Серверна шафа	
Рис. 1.11. Елементи професійних комп’ютерів		

Рис. 1.12. Кластер

Квантовий комп’ютер. Перший комерційний квантовий комп’ютер – 20 кубіт

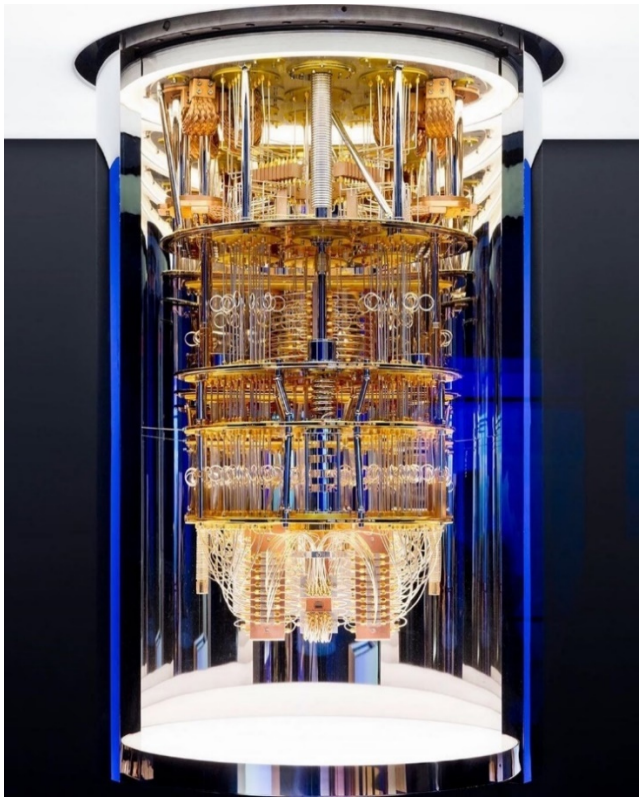


Рис. 1.13. IBM Quantum System One

Одиниця інформації – 1 кубіт (квантовий біт)

Значення кубіта – суперпозиція квантових станів $|0\rangle$ і $|1\rangle$

1.2. Програмне забезпечення комп’ютерів

1.2.1. Архітектура комп’ютера

Архітектуру комп'ютера визначають такі принципи:

1. Принцип програмного керування. Забезпечує автоматизацію процесу обчислень на комп'ютері. Згідно з цим принципом, для розв'язування кожної задачі складається програма, яка визначає послідовність дій комп'ютера.

2. Принцип програми, що зберігається в пам'яті. Згідно із цим принципом, команди програми подаються, як і дані, у вигляді чисел і обробляються так само, як і числа, а сама програма перед виконанням завантажується в оперативну пам'ять, що прискорює процес її виконання.

3. Принцип довільного доступу до пам'яті. Відповідно до цього принципу, програми й дані можуть записуватися в довільне місце оперативної пам’яті, що дозволяє звертатися за будь-якою заданою адресою (до конкретної ділянки пам'яті) без перегляду попередніх.

Комп'ютер – технічне обладнання для обробки даних

1. Введення в пам’ять початкових даних
2. Введення в пам’ять комп'ютерної програми
3. Обробка даних відповідно до програми
4. Вивід результатів розв’язування задачі у формі, придатній для сприйняття людиною

1.2.2. Загальне поняття про програму

Комп’ютерна програма – послідовність інструкцій, що визначають процедуру розв'язування конкретної задачі комп’ютером.

Задача може бути вирішена успішно, якщо програма налагоджена.

Програмне забезпечення (software) – сукупність програм, процедур і правил, а також документації, що стосуються функціонування комп'ютера.

Комп’ютер включає **HARDWARE** та **SOFTWARE**

Software нерозривно пов'язане й функціонує разом з hardware.

Рис. 1.14. Три складові, які необхідні для функціонування комп’ютера

1.2.3. Ієрархія елементів програмного забезпечення

Існує ієрархія різних елементів програмного забезпечення, що утворюють програмну конфігурацію.

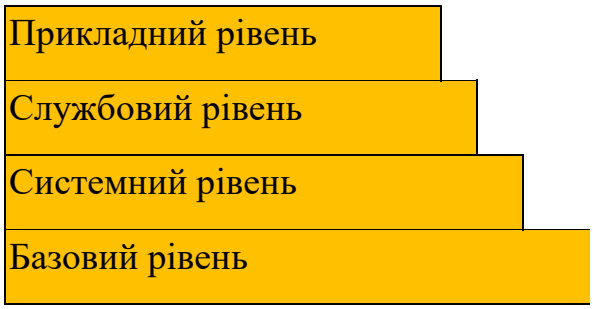


Рис. 1.15. Ієрархія рівнів програмного забезпечення

1.2.4. Базовий рівень програмного забезпечення

Базовий рівень є найнижчим рівнем програмного забезпечення. Відповідає за взаємодію з базовими апаратними засобами.

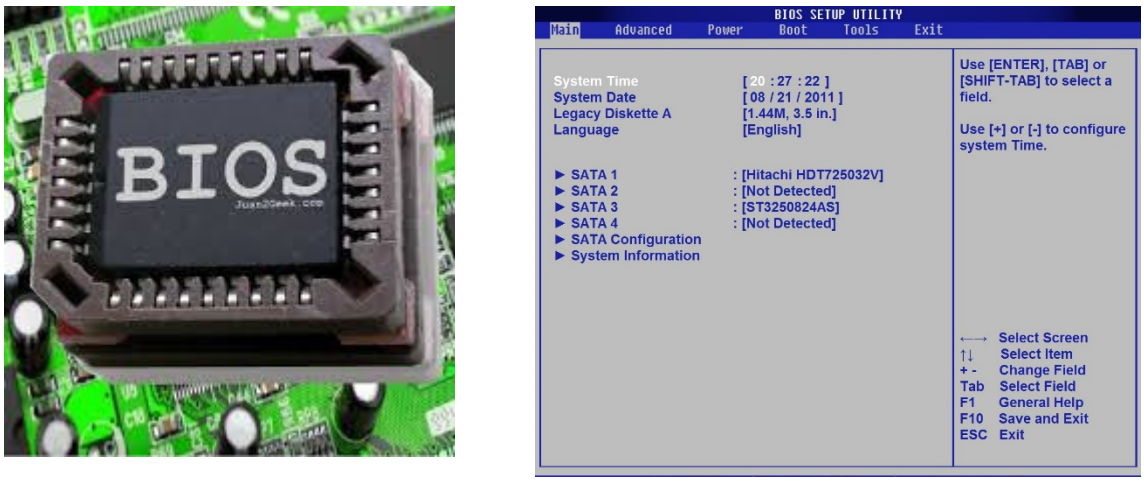


Рис. 1.16. Фізичне розміщення та інтерфейс BIOS

Базове програмне забезпечення міститься в складі базового апаратного забезпечення й зберігається в спеціальних мікросхемах постійного запам'ятовувального пристрою (ROM), утворюючи базову систему вводу-виводу BIOS.

Програми й дані записуються в ROM на етапі виробництва й можуть бути змінені під час експлуатації тільки за певних строго обумовлених обставин.

1.2.5. Системний рівень програмного забезпечення

Системний рівень – є перехідним і складається з трьох груп програм, які разом названі операційною системою комп'ютера (ОС).

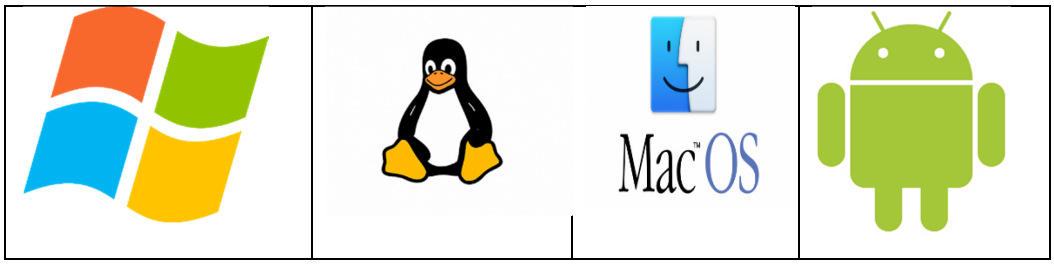


Рис. 1.17. Відомі операційні системи

За всю історію розвитку комп'ютерів було дуже багато різних операційних систем, які відповідали апаратному забезпеченню комп'ютерів.

ОС фірми Microsoft:Windows 3.1, Windows 95, Windows 98,

Windows NT, Windows 2000, Windows XP, Windows 7-11

Unix – подібні ОС

Linux: RedHat, Ubuntu, Debian

FreeBSD,Solaris,MacOS

Групи програм, що входять у ОС

Програми, що взаємодіють з устаткуванням:

Забезпечують взаємодію інших програм з програмами базового рівня й безпосередньо з апаратним забезпеченням.

Рис. 1.18. Програма установки драйверів

При приєднанні до комп'ютера нового обладнання на системному рівні повинна бути встановлена програма, що забезпечує для інших програм взаємозв'язок з обладнанням.

Конкретні програми, призначені для взаємодії з конкретними обладнаннями, називають драйверами.

Програми, що взаємодіють з користувачем:

Відповідають за взаємодію з користувачем і забезпечують:

1.Введення даних у комп'ютер

2.Можливість керувати роботою комп'ютера

3.Одержання результату в зручній формі

Це засоби забезпечення користувацького інтерфейсу, від них залежить зручність і продуктивність роботи з комп'ютером.

Рис. 1.19. Десктоп операційної системи

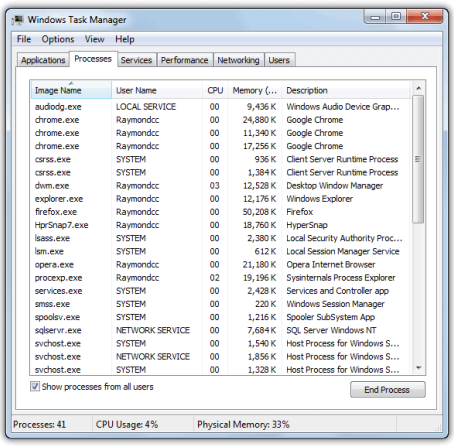


Рис. 1.20. Менеджер задач

Ядро́ (CORE) — центральна частина операційної системи (ОС), що забезпечує прикладним програмам координований доступ до ресурсів комп’ютера, таким як

- процесорний час,
- пам’ять,
- зовнішнє апаратне забезпечення,
- зовнішні пристрої вводу й виводу інформації.

Також зазвичай ядро надає сервіси

файлової системи

і мережних протоколів.

1.2.6. Службовий рівень (утиліти)

Призначення: автоматизація робіт з перевірки й настроювання комп'ютерної системи,

поліпшення функцій системних програм.

Деякі службові програми (програми обслуговування) відразу входять до складу операційної системи, доповнюючи її ядро.

Більшість є зовнішніми програмами й розширюють функції операційної системи. Тобто, у розробці службових програм відслідковуються два напрямки: інтегрРація з операційною системою й автономне функціонування.

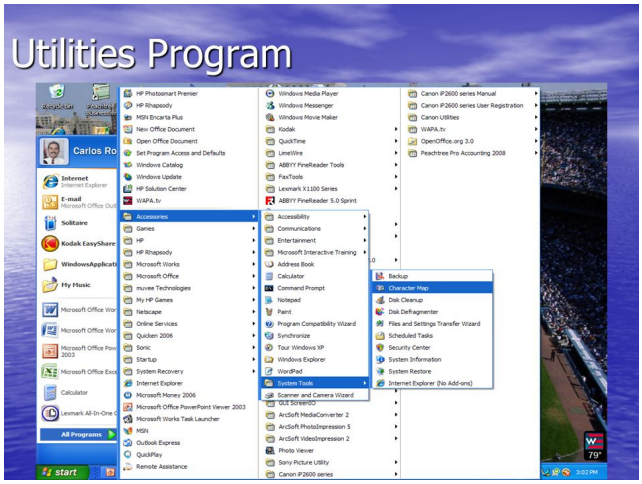


Рис.1.21. Виклик утиліт

Класифікація службових програмних засобів:

1. **Диспетчери файлів** (файлові менеджери). За їх допомоги виконується більшість операцій з обслуговування файлової структури:

копіювання, переміщення,

перейменування файлів,

створення каталогів (папок), знищення об'єктів,

пошук файлів і навігація у файловій структурі.

Приклад відомих диспетчерів файлів:

Назва	**Режим використання**
Explorer	інтегрований у Windows
Norton Commander Total Commander	установлюються окремо

2. **Засоби стискання даних** (архіватори)

Призначені для створення архівів.

Архівні файли мають підвищену щільність запису інформації й, відповідно, ефективніше використовують носії інформації.

WinRAR

Рис. 1.22. Архіватор

2. **Засоби діагностики**

Призначені для автоматизації процесів діагностики програмного й апаратного забезпечення. Їх використовують для виправлення помилок і для оптимізації роботи комп'ютерної системи.

Рис. 1.23. Програма діагностики

4. Програми інсталяції (установки).

Призначені для контролю над додаванням у поточну програмну конфігурацію нового програмного забезпечення. Вони стежать за станом і зміною навколишнього програмного середовища, відслідковують і протоколюють створення нових зв'язків, загублених під час знищення певних програм. Прості засоби керування встановленням і знищенням програм містяться в складі операційної системи, але можуть використовуватися й додаткові службові програми.

Рис. 1.24. Програма інсталяції

5. **Засоби комунікації**

Дозволяють установлювати з'єднання з віддаленими комп'ютерами, передають повідомлення електронної пошти, пересилають факсимільні повідомлення й т. п

.Рис. 1.25. Програма «Telegram»

6. **Засоби перегляду й відтворення**. Переважно, для роботи з файлами, їх необхідно завантажити в "рідну" прикладну програму й ввести необхідні виправлення. Але, якщо редагування не потрібно, існують універсальні засоби для перегляду (у випадку тексту) або відтворення (у випадку звуку або відео) даних.

Рис. 1.26. Приклад програми перегляду даних

7. **Засоби комп'ютерної безпеки**. Це засоби пасивного й активного захисту даних від ушкодження, несанкціонованого доступу, перегляду й зміни даних. Засоби пасивного захисту – це службові програми, призначені для резервного копіювання. Засоби активного захисту застосовують антивірусне програмне забезпечення. Для захисту даних від несанкціонованого доступу, їх перегляду й зміни використовують спеціальні системи, що базуються на криптографії.

Рис. 1.17. Приклади засобів комп’ютерної безпеки

1.2.7. Прикладний рівень програмного забезпечення

Призначення: виконують конкретні задачі:

виробничі – при виробництві продукції,

творчі – при створенні творів мистецтва,

розважальні – комп’ютерні ігри,

навчальні – навчальні програми, тестування.

Класифікація прикладного програмного забезпечення:

1. Текстові редактори
2. Графічні редактори
3. Системи керування базами даних (СУБД)
4. Електронні таблиці
5. Системи автоматизованого проектування (Cad-системи)
6. Редактори HTML (Web-редактори)
7. Браузери
8. Системи автоматизованого перекладу
9. Інтегровані системи діловодства
10. Бухгалтерські системи
13. Фінансові аналітичні системи
14. Експертні системи

15. Геоінформаційні системи (ГІС)

16. Системи відеомонтажу

Алгоритмічні мови програмування:

Алгоритмічна мова — формальна мова, використовувана для запису, реалізації або вивчення алгоритмів.

Алгоритмічна мова програмування — формальна мова програмування, призначена для повідомлення інструкцій комп'ютеру.

Мови програмування можуть бути використані для створення програм керування поведінкою комп'ютера або для запису алгоритмів у вигляді програм певною мовою.

Опис мови програмування, як правило, поділяється на дві складові:

синтаксис (форма запису інструкцій) і

семантика (зміст, вкладений у синтаксичну конструкцію).

Системи програмування

Системи програмування – **засоби розробки програм.**

У них входять:

- Транслятор: компілятор або інтерпретатор;
- Бібліотеки стандартних програм і функцій;
- Графічні бібліотеки; утиліти для роботи з бібліотеками
- вбудований асемблер;
- Інтегроване середовище розробки (IDE):
  - засоби створення й редагування текстів програм;
  - налагоджувальні програми, тобто програми, що допомагають знаходити й усувати помилки в програмі;
  - "дружнє" до користувача діалогове середовище;
  - багатовіконний режим роботи;
  - вбудована довідкова служба;

Транслятор

Трансля́тор — програма, що виконує *трансляцію програми*.

Трансля́ція програ́ми — перетворення програми, представленої однією з мов програмування, в *об'єктний файл.* Транслятор зазвичай виконує також діагностику помилок, формує словники ідентифікаторів, видає для друку текст програми і т. д.

Мову, якою представлена вхідна програма, називають *вхідною мовою*, а саму програму — *вхідним кодом* (початковим кодом).

Вихідну мову називають *цільовою мовою, а вихідну (результуючу) програму — об'єктним кодом*.

Транслятори бувають двох типів:

- інтерпретатори,
- компілятори.

Інтерпретатор читає один оператор програми, аналізує його й відразу виконує,

після чого переходить до очікування введення наступного оператора.

Інтерпрета́ція — пооператорний (покомандний) аналіз, обробка й відразу виконання вхідної програми або запиту (на відміну від компіляції, при якій програма транслюється без її виконання).

Компілятор спочатку читає, аналізує й перекладає на машинний код всю програму й тільки після завершення всієї трансляції ця програма виконується.

Компіля́ція — трансляція програми, складеної вхідною мовою високого рівня, в еквівалентну програму на низькорівневій мові, близькій до машинного коду (абсолютний код, об'єктний модуль, іноді на мову асемблера).

Вхідною інформацією для компілятора (вхідний код) є опис алгоритму або програма на предметно-орієнтованій мові, а на виході компілятора —

еквівалентний опис алгоритму машинно-орієнтованою мовою (об'єктний код).

Рівні мов програмування

Алгоритмічні мови діляться на мови низького рівня (близькі до машинної мови) і мови високого рівня (близькі до людських мов).

Мови низького рівня

Загальновідомий приклад низькорівневої мови — мова асемблера, хоча вірніше говорити про групу мов асемблера.

Є різні мови асемблера для різних процесорів, оскільки вони орієнтовані на архітектуру процесора.

Більше того, для одного і того самого процесора існує кілька видів мови асемблера.

Мови асемблера збігаються в машинних командах, але різняться набором додаткових функцій (директив і макросів).

Мови високого рівня

Існують сотні мов програмування високого рівня. Вони можуть бути спеціалізовані, універсальні й проблемно-орієнтовані.

Методи виміру популярності мови:

- підрахунок кількості оголошень про вакансії, які згадують мову,
- підрахунок кількості проданих книг, які призначені для вивчення або опису мови,
- оцінка кількості існуючих рядків коду, написаного мовою,
- підрахунок посилань на назву мови, знайдених за допомогою веб-пошуку.

Індекс TIOBE (https://www.tiobe.com/tiobe-index/)

Індекс TIOBE обновляється один раз на місяць. Рейтинги засновані на кількості: кваліфікованих інженерів у всьому світі, курсів, рядків коду і т.д. Популярні пошукові системи, такі як Google, Bing, Yahoo!, Wikipedia,

Amazon, Youtube і Baidu використовуються для розрахунків оцінок.

Aug 2021	Aug 2020	Change	Programming Language	Ratings	Change
1	1		C	12.57%	-4.41%
2	3	↑	Python	11.86%	+2.17%
3	2	↓	Java	10.43%	-4.00%
4	4		C++	7.36%	+0.52%
5	5		C#	5.14%	+0.46%
6	6		Visual Basic	4.67%	+0.01%
7	7		JavaScript	2.95%	+0.07%
8	9	↑	PHP	2.19%	-0.05%
9	14	⇈	Assembly language	2.03%	+0.99%
10	10		SQL	1.47%	+0.02%

Історія мов програмування

В таблиці показано середньорічні показники популярності мов програмування, починаючи з 1985 року

Мова	**2021**	**2020**	**2015**	**2010**	**2005**	**2000**	**1995**	**1990**	**1985**
Java	3	1	2	1	2	3	-	-	-
C	1	2	1	2	1	1	2	1	1
Python	2	3	7	6	6	22	20	-	-
C++	4	4	3	4	3	2	1	2	9
C#	5	5	5	5	9	9	-	-	-
JavaScript	7	6	8	8	10	6	-	-	-
PHP	8	7	6	3	5	21	-	-	-
SQL	10	8	-	-	-	-	-	-	-
Swift	16	9	16	-	-	-	-	-	-
R	14	10	12	54	-	-	-	-	-

Index PYPL (PopularitY of Programming Language **)**

(https://pypl.github.io/PYPL.html?country=US) серпень 2021

Rank	Change	Langeage	Share	Trend
1		Python	31.47 %	-4.1 %
2		Java	19.14 %	+3.2 %
3		Javascript	7.49 %	-0.4 %
4	↑	C#	6.24 %	+0.1 %
5	↓	R	5.87 %	+0.5 %
6		C/C++	5.56 %	+1.0 %
7		Objective-C	3.29 %	-0.1 %
8	↑	PHP	2.84 %	-0.3 %
9	↓	Swift	2.27 %	-1.6 %
10	↑↑	Go	2.17 %	+0.5 %

Чим більший пошук підручників з даної мови, тим популярнішою вважається мова. Це провідний показник. Необхідні дані надходять з Google Trends. Автор рейтингу: П’єр Карбоньєл

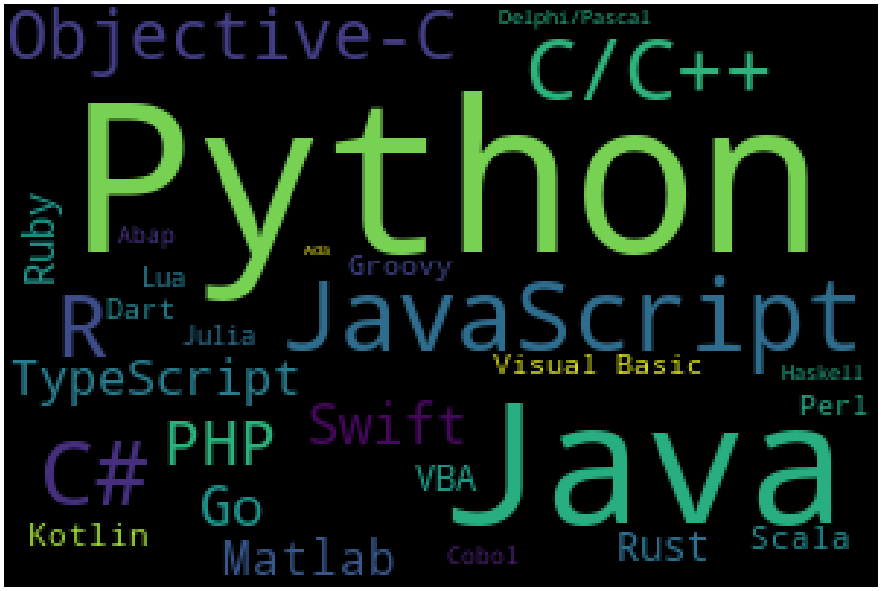


Рис. 1.18. Wordcloud за рейтингом Index PYPL

Інтегроване середовище розробки - IDE

(англ. *Integrated development environment*) — комплекс програмних засобів, використовуваний програмістами для розробки програмного забезпечення (ПЗ).

IDE створені для того, щоб максимізувати продуктивність програміста завдяки тісно пов'язаним компонентам з простими користувацькими інтерфейсами.

Це дозволяє розроблювачеві зробити менше дій для перемикання різних режимів, на відміну від дискретних програм розробки.

Однак оскільки IDE є складним програмним комплексом, то середовище розробки зможе якісно прискорити процес розробки ПЗ лише після спеціального навчання.

Для зменшення бар'єра входження в програмні системи багато IDE підтримують різні мови програмування.

IDE зазвичай є єдиною програмою, у якій проводиться вся розробка. Вона, як правило, містить багато функцій для створення,
зміни,
компілювання,
розгортання й
налагодження програмного забезпечення.

Мета інтегрованого середовища полягає в тому, щоб об'єднати різні утиліти в одному модулі, який дозволить абстрагуватися від виконання допоміжних задач, тим самим дозволить програмістові зосередитися на розв'язуванні власне алгоритмічної задачі й уникнути втрат часу при виконанні типових технічних дій (наприклад, виклику компілятора).

Таким чином підвищується продуктивність праці розробника.

IDE містить у собі:

- Текстовий редактор. (Editor)

- Іноді компілятор і/або інтерпретатор.(Compiler)
- Налагоджувач. (Debugger)

Те́кстовий реда́ктор - компонент програмного комплексу призначений для створення та редагування текстових даних в інтерактивному режимі.

(наприклад, редактор вхідного коду IDE),

Текстові редактори дозволяють виконувати дії над файлами:

- переглядати вміст,
- робити вставку, видалення й копіювання тексту,

виконувати контекстний пошук і заміну, сортування рядків,

- конвертацію кодувань, друк і т. ін.

Додаткові функції текстового редактора

- запам'ятовування послідовності натиснутих клавіш,
- вбудовані підказки по конкретній мові програмування,
- відображення текстових даних спеціальним чином з підсвічуванням синтаксису.

Налагоджувач (**деба́ггер**, англ. *debugger* від *bug*) — комп'ютерна програма, призначена для пошуку помилок у програмах.

Налагоджувач дозволяє:

-виконувати трасування,

-відслідковувати, установлювати або змінювати значення змінних у процесі виконання коду,

-установлювати й видаляти контрольні точки або умови зупинки і т. д.

1.3. Початкові поняття про мову програмування Python

1.3.1. Основні характеристики мови Python

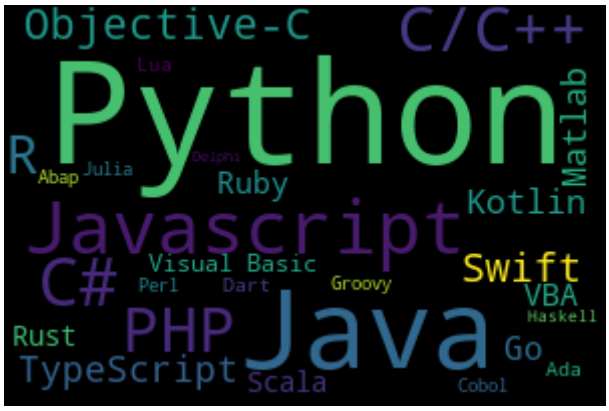


Рис. 1.19 WordCloud популярності мови Python

1. Курс присвячено одній з популярних мов програмування, що бурхливо розвиваються – Python. Мова Python дозволяє швидко створювати програми, допомагає в інтеграції програмного забезпечення для розв'язування виробничих завдань.

Рис. 1.20. Наявність бібліотек

2. Python має багату стандартну бібліотеку й велику кількість модулів розширення практично для всіх потреб у галузі інформаційних технологій.

Рис. 1.21. Широке поширення мови

3. Завдяки зрозумілому синтаксису вивчення мови не становить великої проблеми. Тому вона має широке розповсюдження.

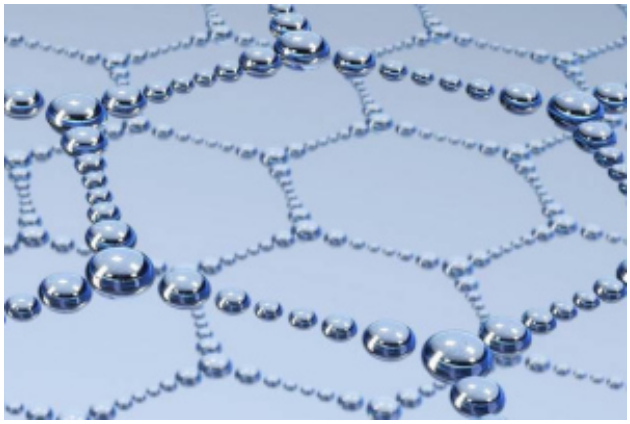


Рис. 1.22. Висока структурованість мови

4. Написані на ній програми структуровані за формою, і в них легко простежити логіку роботи.

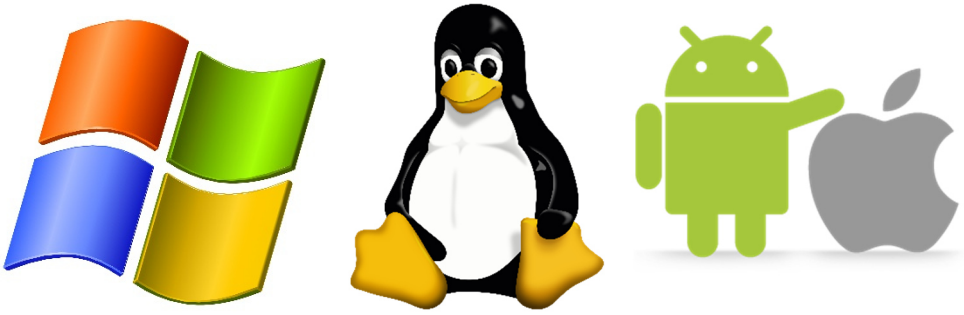
Рис. 1.23. Об’єктно орієнтоване програмування

5. На прикладі мови Python легко усвідомити такі важливі поняття як:

- об'єктно-орієнтоване програмування (ООП),
- функціональне програмування,
- подійно-керовані програми (Gui-додатки),
- формати представлення даних (Unicode, XML і т. п.).

Рис. 1.24. Наявність інтерпретатора

6. Можливість діалогового режиму роботи інтерпретатора Python дозволяє істотно скоротити час вивчення мови.



Windows Linux Android ios

7. Python вільно доступний і швидко розвивається для багатьох платформ, а написані на ньому програми зазвичай переносяться між платформами без змін.

1.3.2. Історія створення та складові мови Python

Створення Python було почато **Гвідо ван Россумом** (Guido van Rossum) у **1991 році**.

Він вибрав назву Python на честь комедійних серій BBC "Літаючий цирк Монти-Пайтона", а зовсім не за назвою змії.

З того часу Python розвивався за підтримки тих організацій, у яких Гвідо працював.

Рис. 1.25. Гвідо ван Россум

Особливо активно мова удосконалюється тоді, коли над нею працює не тільки команда творців, а й співтовариство програмістів усього світу. Однак

останнє слово про напрямок розвитку мови залишається за Гвідо ван Россумом.

Як вимовляється назва мови

Рис. 1.26. Походження назви мови Python

Python краще вимовляти "пайтон", оскільки це англійське звучання назви мови поширено в усьому світі, хоча деякі говорять "пітон".

Python – це універсальна мова програмування. Вона має свої переваги й недоліки, а також сфери застосування.

1. Універсальні бібліотеки.
2. Бібліотеки з різних предметних областей:
   - засоби обробки текстів і технології Інтернет,
   - обробка зображень,
   - інструменти для створення додатків,
   - механізми доступу до баз даних,
   - пакети для наукових обчислень,
   - бібліотеки побудови графічного інтерфейсу і т. п.
3. Засоби інтеграції з мовами C, C++ (і Java)

Підтримує 3 парадигми програмування

Мова Python підтримує три основні **парадигми** програмування:

1. імперативне програмування
   (процедурний, структурний, модульний підходи),

2. об'єктно-орієнтоване програмування,
3. функціональне програмування.

Python – це технологія для створення програмних продуктів. Вона доступна майже на всіх сучасних платформах (як 32-бітних, так і на 64-бітних)

Багато хто вважає, що немає кращого, ніж C/C++, Java, Visual Basic, C#. Однак це не так. Завдяки даному курсу в Python обов'язково з'являться нові прихильники, для яких він стане незамінним інструментом.

Технологія освоєння мови Python

Для вивчення деталей опису ви повинні використовувати інструкцію користувача, інші допоміжні методичні матеріали, книги.

Тим більше, що деякі версії мови можуть мати відмінності.

Ми будемо вивчати технологію програмування мовою Python, використовуючи велику кількість прикладів і практичних завдань.

Мета такого підходу – забезпечити відсутність збоїв у ланцюжку **семантика -> синтаксис -> прагматика.**

семантика -> синтаксис -> прагматика

Семантика – це той «зміст», який програміст вкладає у свою програму.

Синтаксис – це засіб, за допомогою якого програміст викладає свій задум у вигляді, зрозумілому інтерпретатору.

Прагматика – це ті дії, які виконує інтерпретатор з реалізації завдання, представленого в синтаксично правильному вигляді.

В цьому є основна перевага Python. Синтаксис мови Python має потужні засоби, які допомагають наблизити розуміння проблеми програмістом до її «розуміння» інтерпретатором. Саме тому Python має найнижчий поріг «входження».

Структура програми Python

Програма на Python складається з рядків. Кожен рядок може бути фізичним або логічним

У одному фізичному рядку можна розмістити до 80 символів

Логічний рядок складається з кількох фізичних рядків. Для створення логічного рядка з фізичних рядків використовують спеціальні символи

Дзен Пайтона

Дзен або «споглядання», одна з найважливіших шкіл китайського і всього східно-азіатського буддизму

Якщо інтерпретатору Пайтона дати команду

import this (імпортувати «сам об'єкт»),

то виведеться так званий «Дзен Пайтона», що ілюструє ідеологію й особливості даної мови.

Глибоке розуміння цього дзену приходить до тих, хто зможе освоїти мову Python повною мірою й набуде досвіду практичного програмування.

1. Beautiful is better than ugly. Гарне краще, ніж потворне.
2. Explicit is better than implicit. Явне краще, ніж неявне.
3. Simple is better than complex. Просте краще, ніж складне.
4. Complex is better than complicated. Складне краще, ніж заплутане.
5. Flat is better than nested. Плоске краще, ніж вкладене.
6. Sparse is better than dense. Розріджене краще, ніж щільне.
7. Readability counts. Легкість для читання (читабельність) має значення.
8. Special cases aren't special enough to break the rules. Особливі випадки не настільки істотні, щоб порушувати правила.
9. Although practicality beats purity. Хоча практичність важливіша за бездоганність.
10. Errors should never pass silently. Помилки ніколи не повинні замовчуватися.
11. Unless explicitly silenced. За винятком замовчування, яке задано явно.
12. In the face of ambiguity, refuse the temptation to guess. У випадку неоднозначності опирайтеся спокусі вгадати.

13. There should be one — and preferably only one — obvious way to do it. Повинен існувати один — і, бажано, тільки один — очевидний спосіб зробити це.
14. Although that way may not be obvious at first unless you're Dutch. Хоча він може бути з першого погляду не очевидний, якщо ти не голландець.
15. Now is better than never. «Зараз» краще, ніж «ніколи».
16. Although never is often better than *right* now. Хоча, «ніколи» інколи (і часто!) краще, ніж «прямо зараз».
17. If the implementation is hard to explain, it's a bad idea. Якщо реалізацію складно пояснити — це погана ідея.
18. If the implementation is easy to explain, it may be a good idea. Якщо реалізацію легко пояснити — це може бути гарна ідея.
19. Namespaces are one honking great idea — let's do more of those! Простори імен — прекрасна ідея, давайте робити їх більше!

Інсталяція мови PYTHON

1. Йдемо за посиланням https://www.python.org/

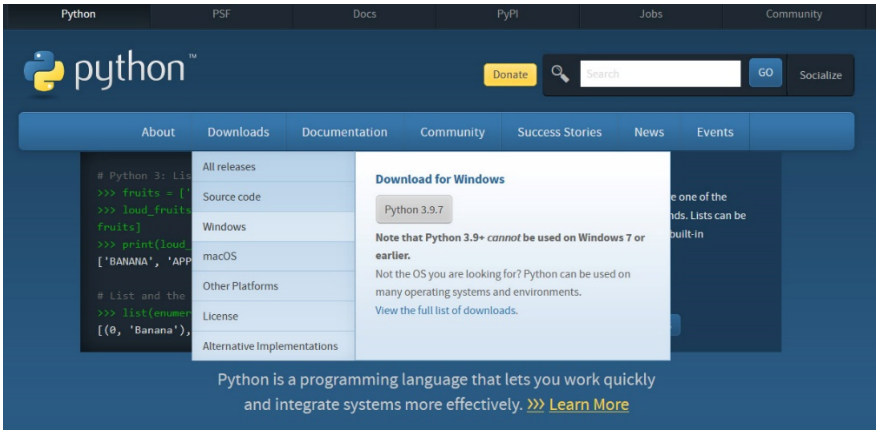


Рис. 1.27. Сайт інсталяції мови Python

2. Скачуємо інсталятор
3. Діємо у відповідності до інструкції на http://amodm.pp.ua

Базове середовище програмування на Python

Середовище для зручного створення програмного коду – IDE (Integrated Development Envoronment).

З Python поставляють найпростішу IDE

Це середовище називають IDLE (читають як Айдл).

Для запуску IDLE зайдіть у меню

Пуск->Python 3.9->IDLE (Python 3.9 64-bit).

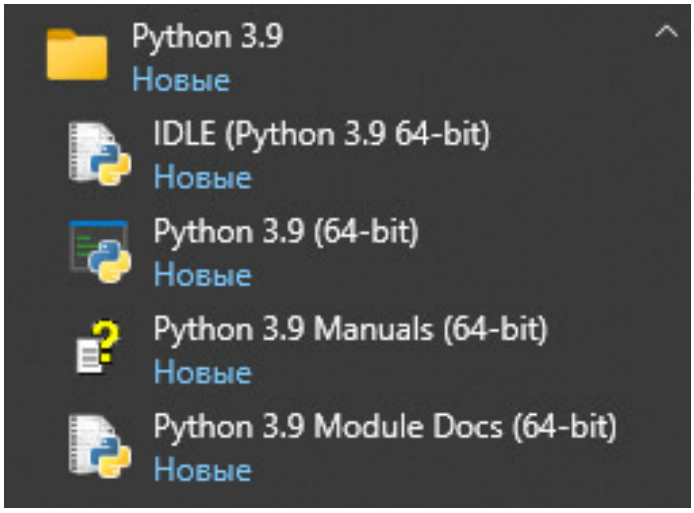


Рис. 1.28. Базове середовище Python

Вхід в програму IDLE

Після запуску програми IDLE відкривається вікно із приблизно таким текстом:

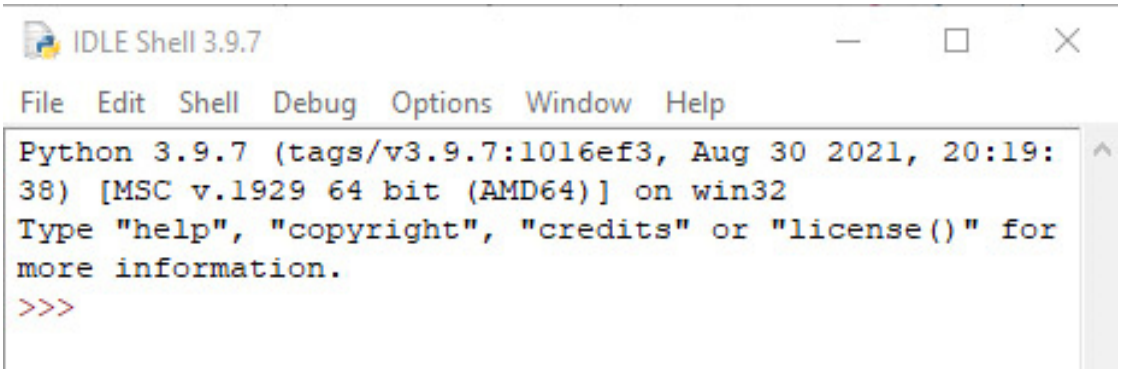


Рис. 1.29. Початкове вікно IDLE

Символ «>>>» – це запрошення інтерпретатора Python до введення команди.

1. Команди вводимо з клавіатури

2. Для виконання команди натискаємо клавішу «Enter»
3. При введені будь якого числа, інтерпретатор виведе його на екран і перейде в режим очікування вводу

Основні правила роботи в середовищі IDLE

IDLE – (Integrated Development and Learning Environment) мови Python. IDLE має наступні особливості:

* Крос-Платформна: працює в основному на Windows, Unix і Mac OS X

* Вікно Python shell (інтерактивний інтерпретатор) з розфарбованим кодом вводу, виводом та повідомленнями про помилки.

* Мультивіконний текстовий редактор з декількома undo, Python розфарбовуванням, смарт-відступами, підказками, що спливають, автозаповненням і іншими функціями

* Відлагоджувач з установлюваними контрольними точками, покроковим режимом, переглядом глобальних і локальних просторів імен

Загальний вигляд Python 3.9.7 Shell

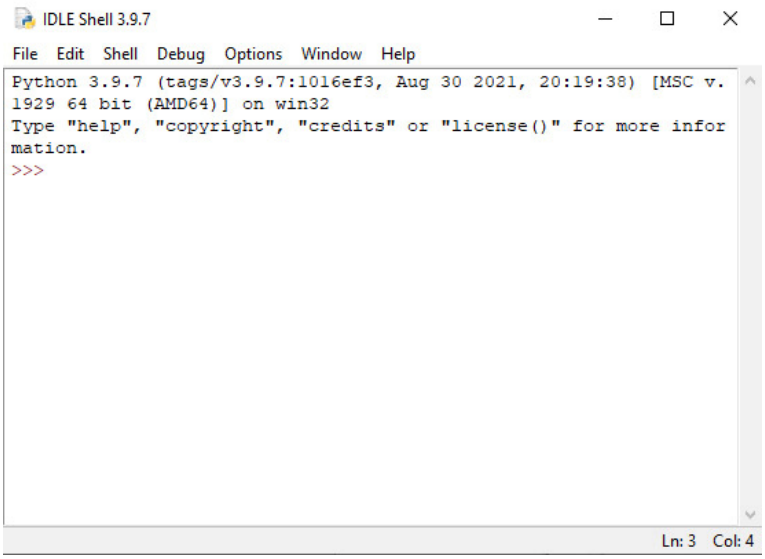


Рис. 1.30. Вікно Shell IDLE

Вікно містить меню та загальне поле інтерпретатора для керування програмою

Інтерпретатор в режимі калькулятора

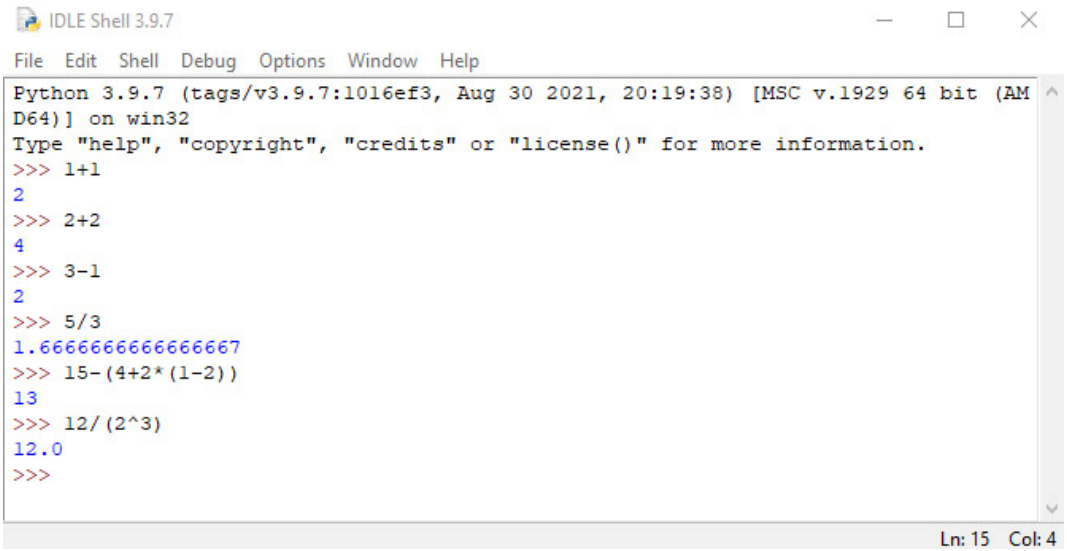


Рис. 1.31. Приклад роботи інтерпретатора

1. Можна виконувати арифметичні дії над числами та записувати арифметичні вирази з цими діями.
2. Можна записувати довільну послідовність символів у лапках або апострофах.

Меню програми IDLE

IDLE має два основні типи вікон:

1.Вікно оболонки: Python Shell

2.Вікно редактора: Edit – name.py-path to file .

Можна мати кілька вікон редактора одночасно.

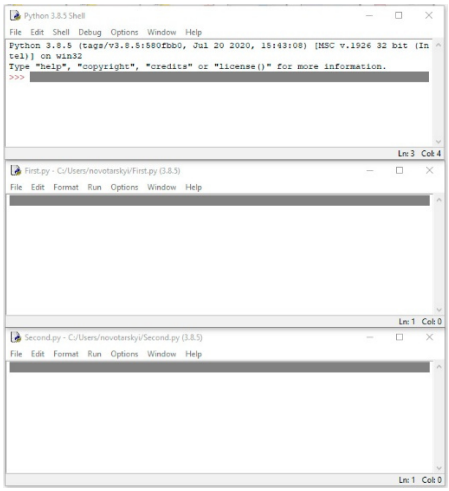


Рис. 1.32. Багатовіконний режим

Меню Файл (Shell і Editor)

Меню IDLE динамічно змінюється залежно від того, яке вікно в цей момент обране. Кожне меню описується нижче. Також вказується на те, який тип вікна з ним зв'язаний

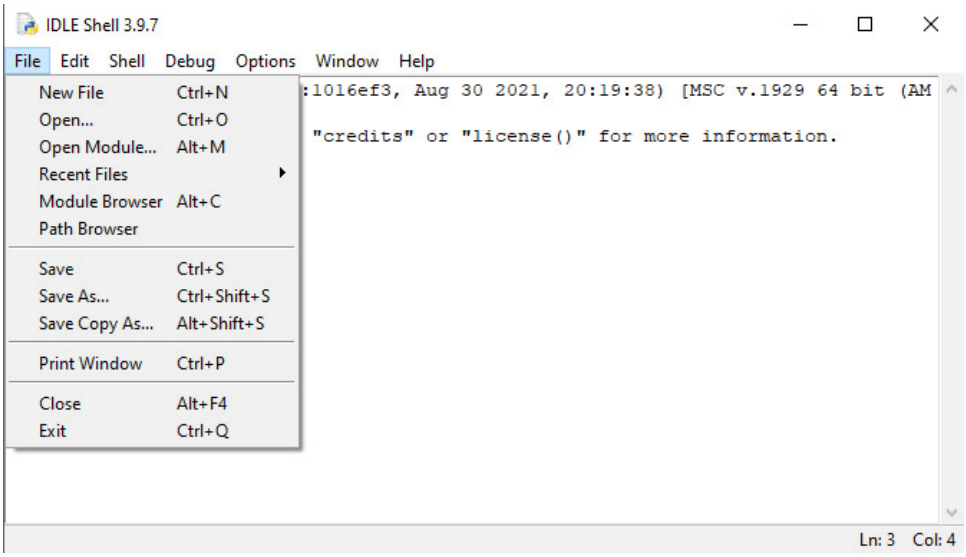


Рис. 1. 33. Меню File

New File

Створити нове вікно для редагування файлу.

Open...

Відкрити існуючий файл за допомогою Open-діалогу.

Recent Files

Відкрити список останніх файлів. Клікніть (натисніть кнопку мишки) один з них, щоб відкрити.

Open Module...

Відкрити існуючий модуль (шукається в sys.path).

Close

Закрити поточне вікно (попросить зберегти, якщо файл незбережений).

Exit

Закриває всі вікна й виходить з IDLE (попросить зберегти незбережені вікна).

Меню Edit (Shell і Editor)

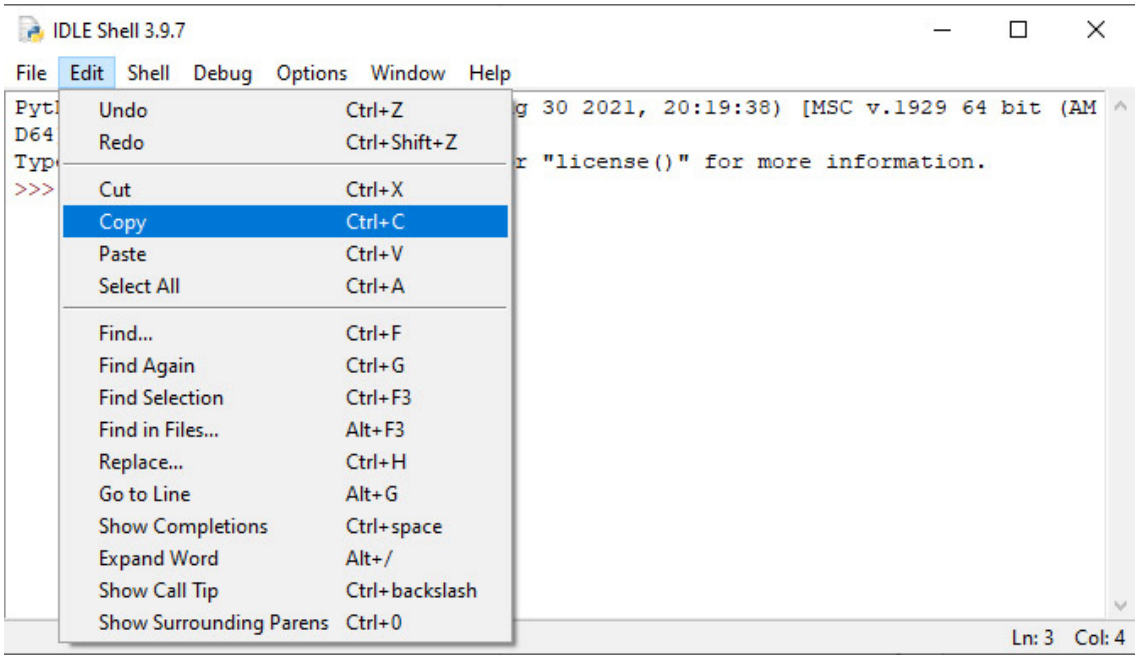


Рис. 1.34. Меню Shell і Editor

Undo Скасовує останню зміну в поточному вікні. Максимум 1000 змін може бути скасовано.

Redo Повернути останню скасовану зміну поточного вікна.

Cut Копіювання виділеного тексту в системний буфер обміну; потім видалення цього виділеного тексту.

Copy Копіювання виділеного тексту в системний буфер обміну без наступного видалення.

Paste Вставити вміст системного буфера обміну в поточне вікно.

Функції буфера обміну також доступні в контекстному меню.

Select All Вибрати весь вміст поточного вікна.

Find...

Відкрийте діалогове вікно пошуку з більшою кількістю опцій

Find Again

Повторити останній пошук, якщо такий є.

Run menu (тільки вікно редактора)

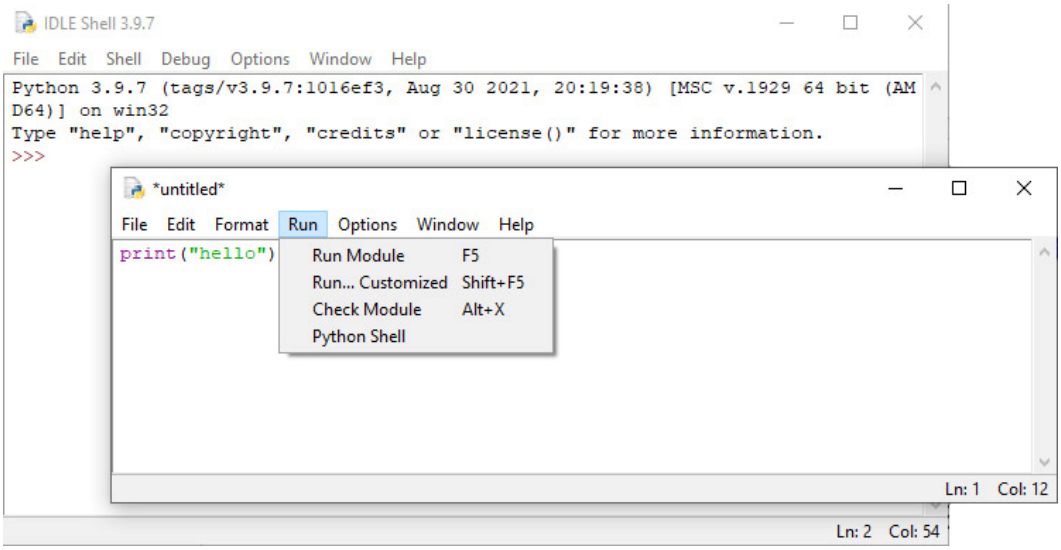


Рис. 1.35 Меню Run

Python Shell Відкриває або активізує вікно оболонки Python.

Check Module Перевіряє синтаксис модуля, відкритого в цей момент у вікні редактора. Якщо модуль не був збережений, IDLE автозберігає файл.

Run Module

Перевіряє модуль, як і в пункті **Check Module.** Якщо немає помилки, то перезапускає оболонку для очищення навколишнього середовища, а потім виконує модуль.

Вихідні дані відображаються у вікні Shell. Зверніть увагу, що вивід даних вимагає використання print або write.

Коли виконання буде завершено, Shell зберігає фокус і відображає підказку. У цей момент можна в інтерактивному режимі досліджувати результат виконання. Це схоже на виконання файлу за допомогою командного рядка python -i file.

1.3.3. Установка програмного забезпечення

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 2021.05 можна за посиланням

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

Anaconda працює під операційними системами Windows, Mac OS X і Linux. Попередньо необхідно вибрати операційну систему. Тоді буде доступна версія Anaconda3 для Python 3.9, як показано на скріншоті:

Необхідно скачати дистрибутив Anaconda3-2021.05-Windows-x86_64.exe і запустити його. Процес інсталяції продемонстрований на скріншотах нижче.

Це вікно одержимо після запуску дистрибутиву:

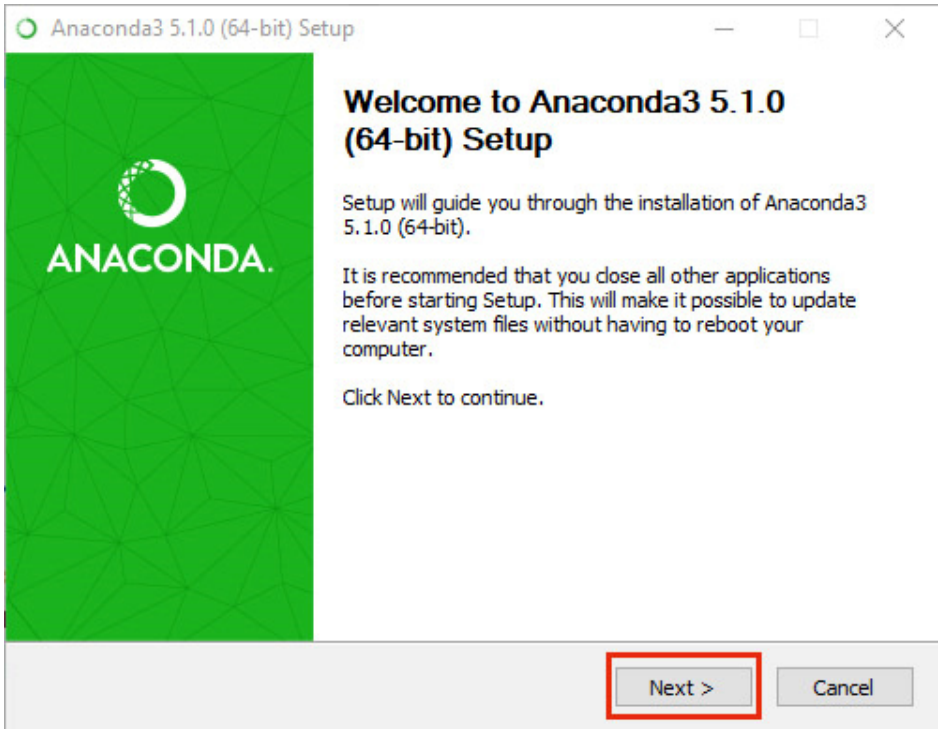


Рис. 1.36. Стартове вікно інсталятора Anaconda

Next натискаємо, впевнившись попередньо, що всі додатки закриті. Наступне вікно – це ліцензійна угода. Далі натискаємо з чистою совістю кнопку «I Agree»:

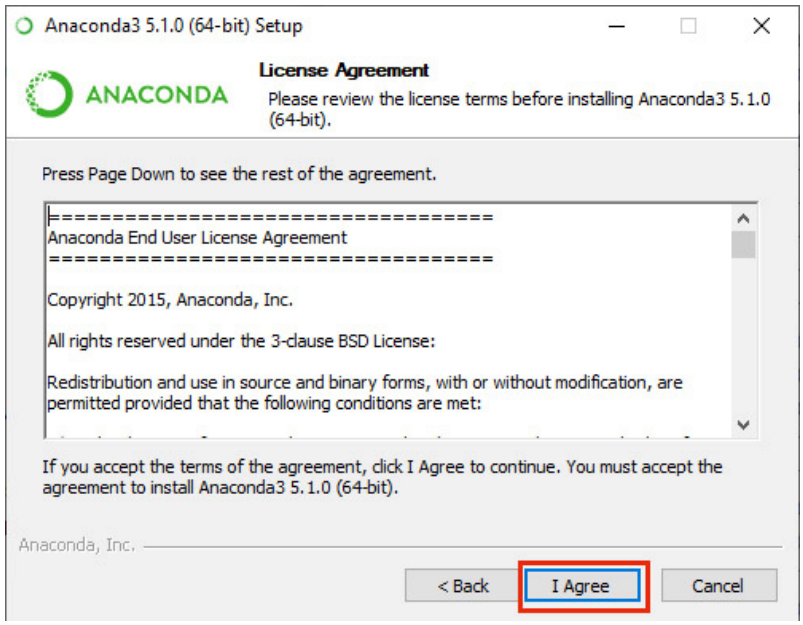


Рис. 1.37. Вікно ліцензування Anaconda

Наступний скріншот – вказує характер доступу. Під Windows можна вибрати All Users, якщо ви маєте права адміністратора. В інших OS вибирайте відповідно до прав доступу.

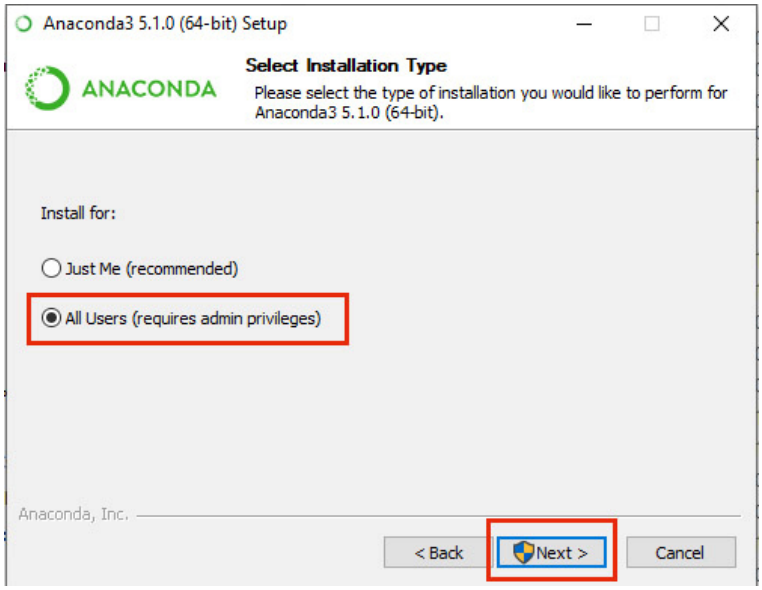


Рис. 1.38 . Вибір рівня доступу до Anaconda

За замовчуванням Anaconda буде інстальована в папку Program Files. Дистрибутив Anaconda можна встановити в попередньо створену папку, наприклад, Anaconda3, як показано на скріншоті.

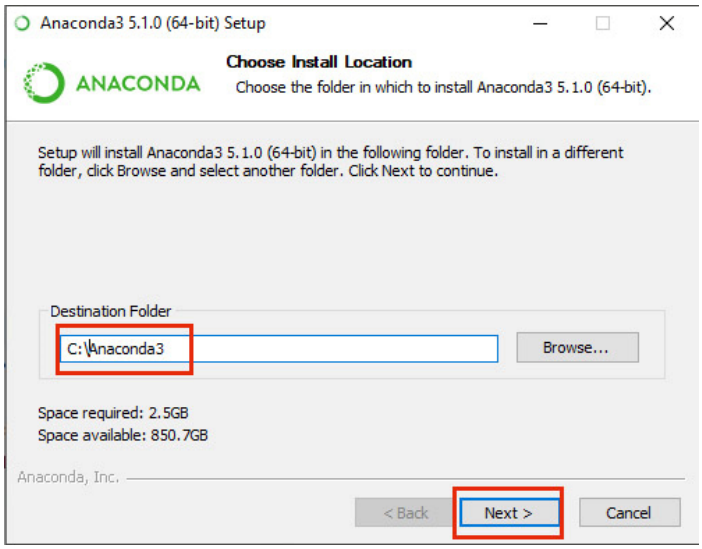


Рис. 1.39. Вибір шляху розміщення Anaconda

Оскільки Python 3.9 попередньо встановлено у папку Program Files, то будемо користуватися саме цим дистрибутивом, вибравши відповідні пункти, як зображено на наступному скріншоті.

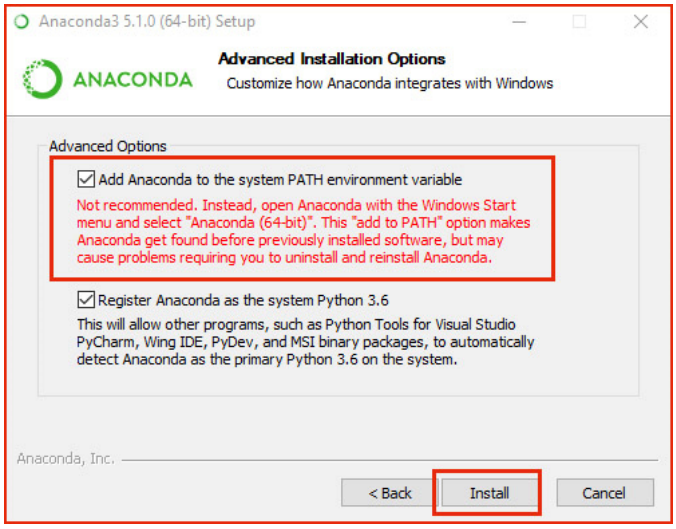


Рис. 1.40. Установка каталогів для запуску

Далі починається процес встановлення Anaconda, який може продовжуватися кілька хвилин в залежності від потужності вашого

комп’ютера. Цей процес відбувається автоматично і не потребує вашого втручання. Процес інсталяції показаний на наступному скріншоті

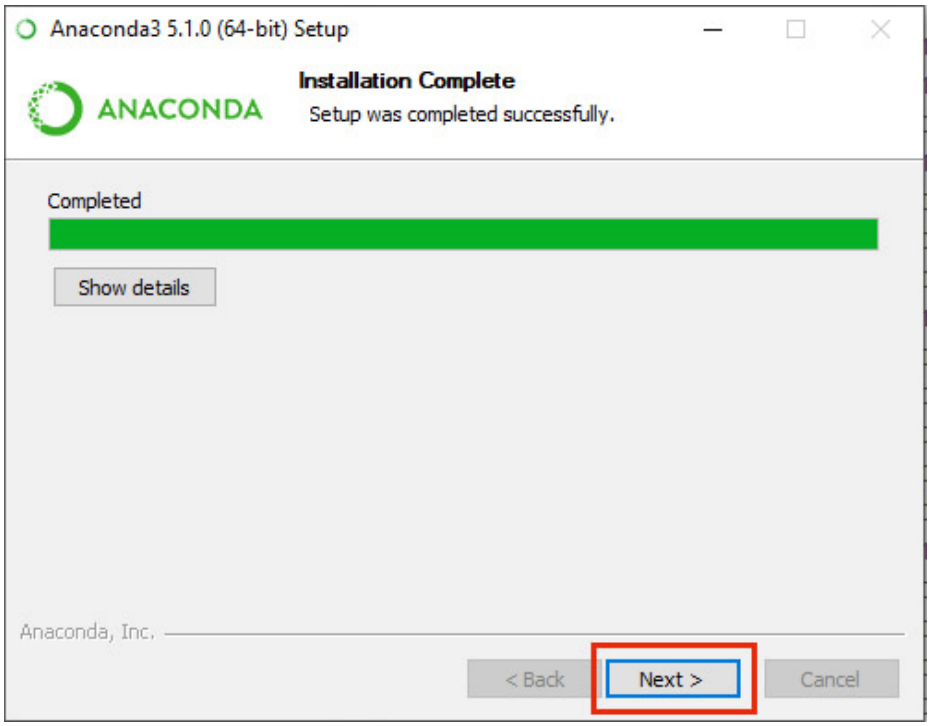


Рис. 1.41. Фінальне вікно інсталяції

У цьому вікні натиснути кнопку Skip.

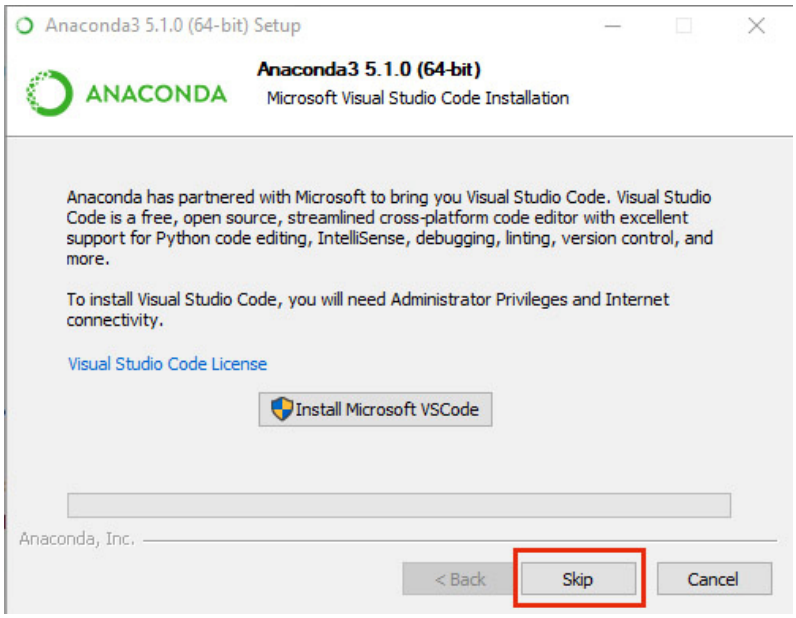


Рис. 1.42. Вікно запуску супутніх застосунків

Заключний екран інсталяції

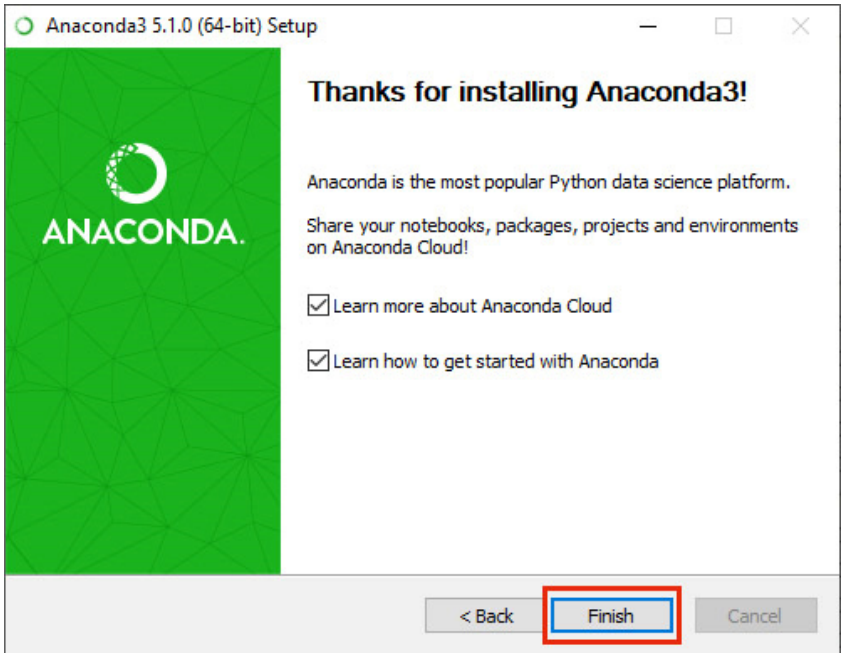


Рис. 1.43. Фінальне вікно інсталяції

1.3.4. Установка PyCharm

PyCharm — інтегроване середовище розробки для мови програмування Python.

Надає засоби для аналізу коду,

графічний редактор коду,

інструмент для запуску юніт-тестів

і підтримує веб-розробку на Django.

PyCharm розроблена чеською компанією JetBrains на основі IntelliJ IDEA.

PyCharm працює під операційними системами Windows, Mac OS X і Linux.

Існують такі версії середовища PyCharm:

PyCharm Professional

для професійної розробки, включаючи Web-Розробку

PyCharm Community

для професійної розробки, крім Web-Розробки

PyCharm Edu

аналогічна Community, але з додатковими можливостями для навчання

Версії PyCharm Community та PyCharm Edu є відкритими та безкоштовними.

Завантаження PyCharm & Anaconda

Процес отримання ліцензії на PyCharm можна почати за посиланням: https://www.jetbrains.com/ru-ru/community/education/#students

Спеціальна версія PyChatm & Anaconda містить вбудований плагін для зв'язку з Anaconda.

Існує два дистрибутиви PyCharm for Anaconda.

PyCharm Community for Anaconda є безкоштовною і знаходиться за посиланням https://download.jetbrains.com/python/pycharm-community-anaconda-2020.2.exe

PyCharm Professional for Anaconda платна, але для викладачів та студентів можна одержати безкоштовну ліцензію. Скачати дистрибутив можна за цим посиланням: https://www.jetbrains.com/lp/pycharm-anaconda/

Порівняльні можливості двох версій показані нижче:

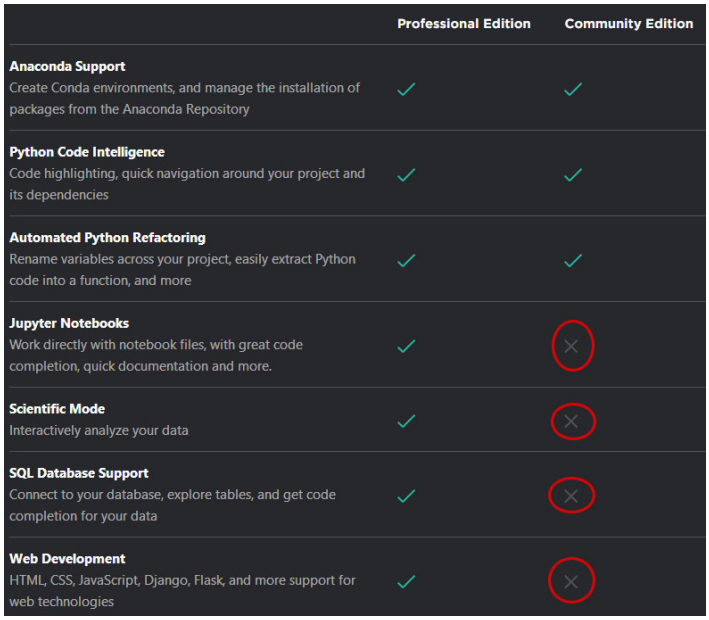


Рис. 1.44. Переваги PyCharm Professional

Важливим пунктом є Jupyter notebooks при роботі у сфері Data Science. Тому бажано отримати ліцензію на версію PyCharm Professional for Anaconda.

Для цього необхідно створити Accaunt на сайті виробника. Для створення Accaunt-а необхідно мати Email у домені КПІ або вислати скан студентського квитка. Для отримання ліцензії почніть з натискання кнопки:

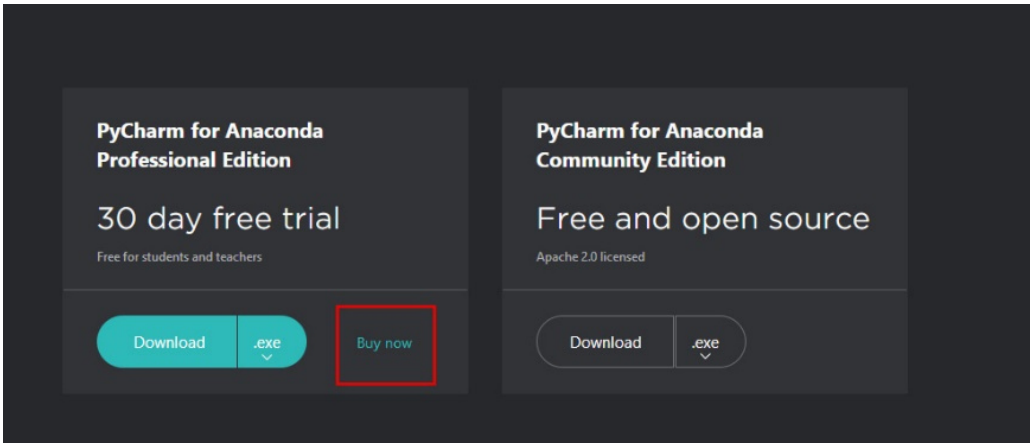


Рис. 1.45. Вікно отримання ліцензії

Після скачування файла pycharm-professional-anaconda-2020.7.exe або pycharm-community-anaconda-2020.7.exe запустити інсталятор.

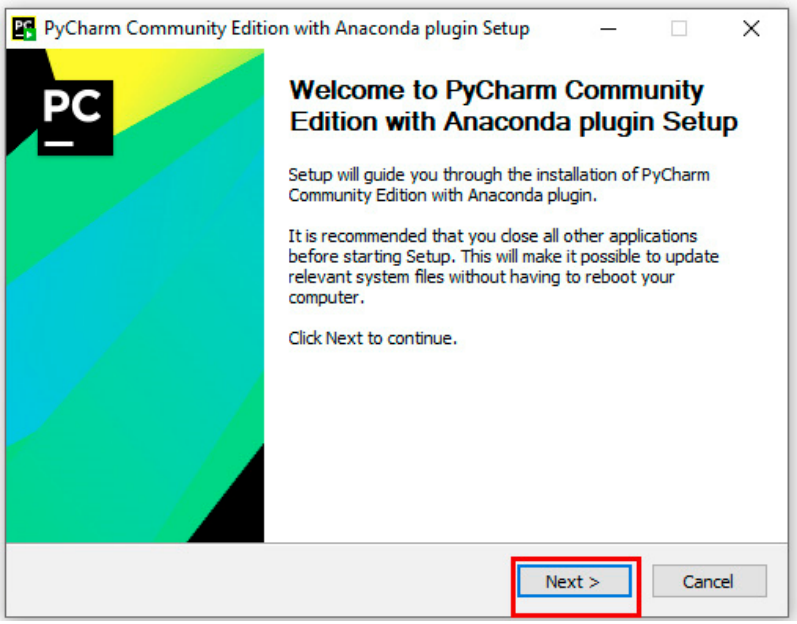


Рис. 1.46. Початкове вікно інсталяції

При інсталяції PyCharm важливо зробити такі налаштування:

2. Друге вікно інсталлятора PyCharm має вигляд:

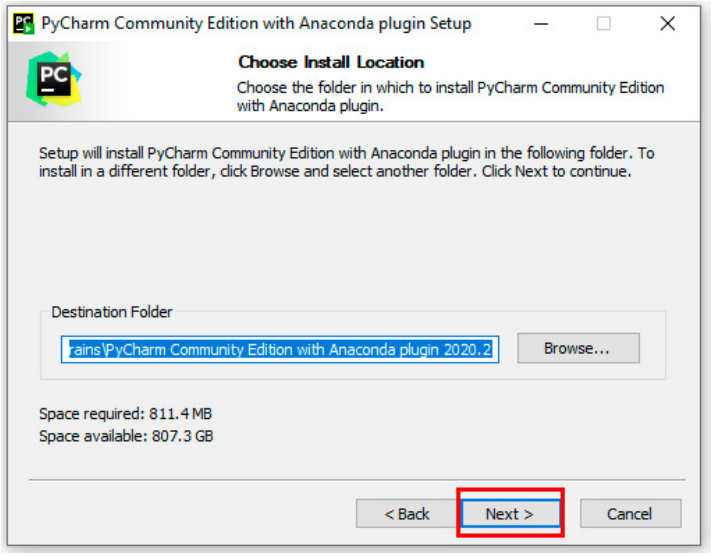


Рис. 1.47. Вікно встановлення шляху установки

У цьому вікні можна встановити шлях до папки, у яку буде встановлено PyCharm. Бажано зберегти цей шлях незмінним. Це позбавить вас проблем з додатковими налаштуваннями шляхів у майбутньому.

3.З наступними налаштуваннями при інсталяції PyCharm треба погоджуватись аж доки не одержите таке вікно:

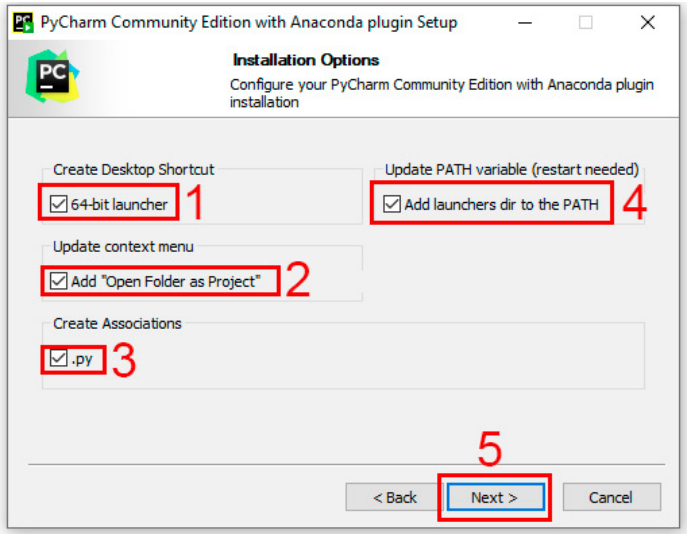


Рис. 1.48. Вікно установки режимів

Послідовність налаштувань у даному вікні показана на скріншоті. Налаштування 2 і 3 покращують зручність роботи з PyCharm, але не є обов'язковими. Налаштування 1 вибираємо, якщо ваша ОС є 64-розрядною. Налаштування 4 автоматично додає посилання на шлях до PyCharm у системне середовище.

3. Далі починається процес інсталяції програмного продукту PyCharm, протяжність якого залежить від потужності вашого комп'ютера.

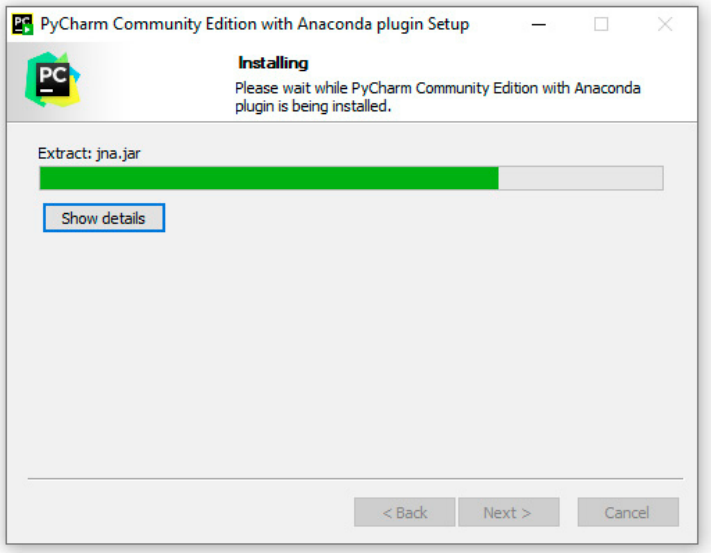


Рис. 1.49. Вікно прогресу установки

5. Фінальне вікно інсталятора PyCharm має вигляд:

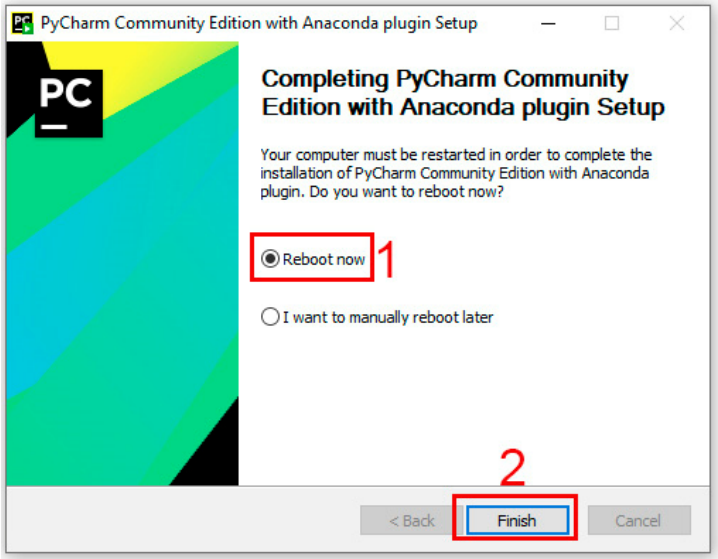


Рис. 1.50. Фінальне вікно

Для правильної роботи PyCharm необхідно перезавантажити комп'ютер. Тому бажано вибрати пункт «Reboot now». Натискання кнопки «Finish» закриває вікно та завершує процес інсталяції.

Початок роботи в IDE PyCharm

6.При першому запуску PyCharm матиме приблизно такий вигляд. Далі необхідно створити свій новий проект, вибравши пункт меню "Create New Project", як показано на скріншоті.

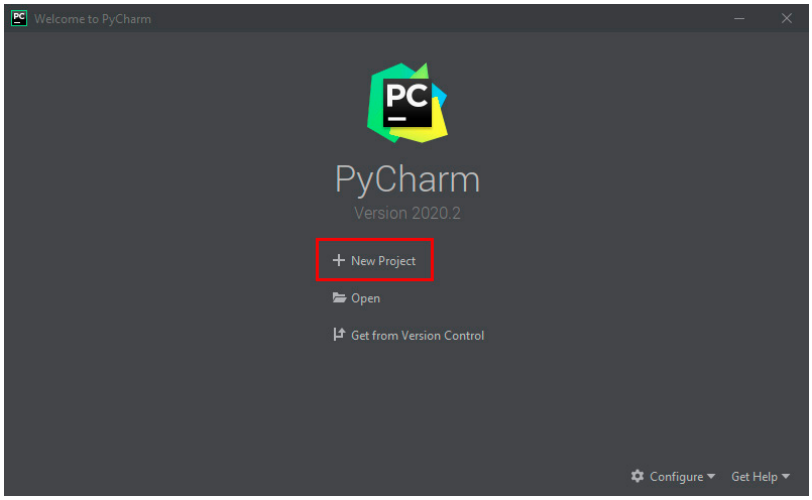


Рис. 1.51 Вікно створення проєкту

7. У першому рядку необхідно вказати шлях до попередньо створеної папки проекту.

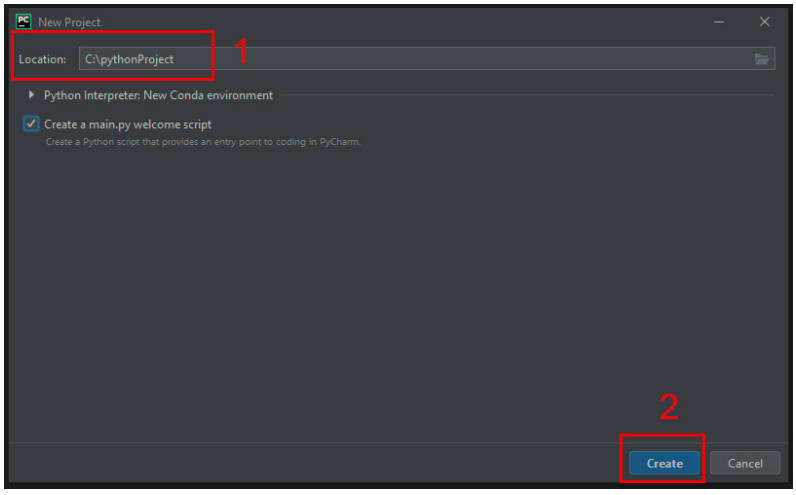


Рис. 1.51 Вікно встановлення шляху до проєкту

Натиснувши кнопку «Create», відкриваємо інтерфейс PyCharm.

8. Отримаємо середовище PyCharm, яке містить вікно проєкту 1 та вікно редактора коду 2.

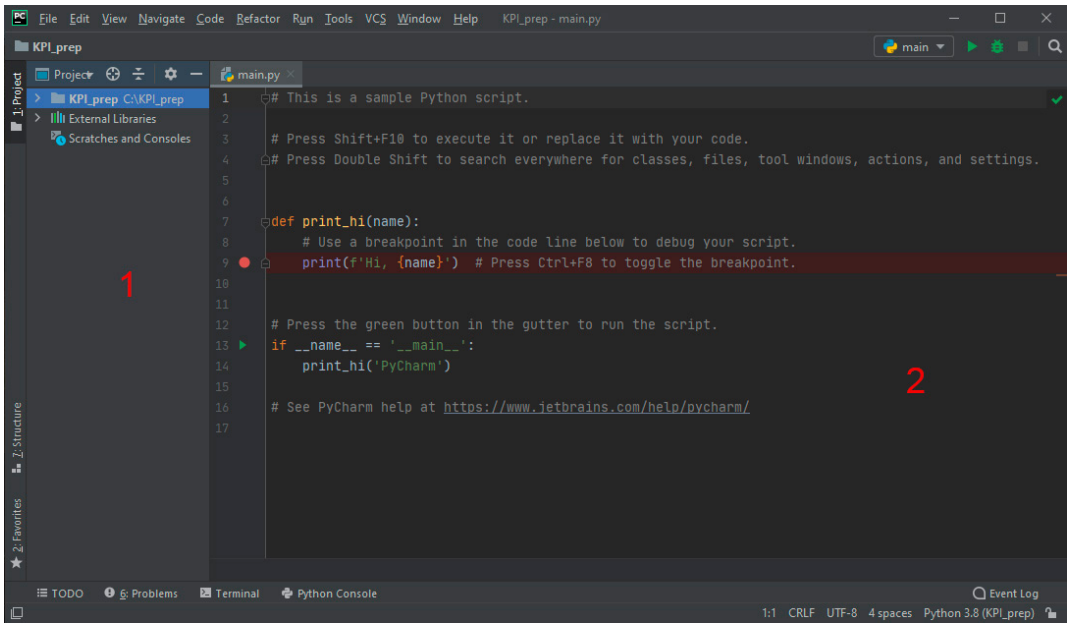


Рис. 1.52. Приклад робочого вікна

У вікні 1 ми можемо переглядати назву папки проїкту та його вміст, а також бібліотеки мови Python. У вікні 2 відкрито файл main.py, який міститьпочатковий приклад. В тексті наведено інструкції по запусу даного коду.

Зміна теми

9.Поточна тема має назву «Dracula» і представлена чорним фоном зі світлими символами. Така тема є неприйнятною для роздруківки фрагментів коду. Для того, щоб змінити тему, вибираємо: File->Settings->Appearanca & Behavior-> Appearance->Theme. Зміна теми на «InteliJ Light» показана на скріншоті:

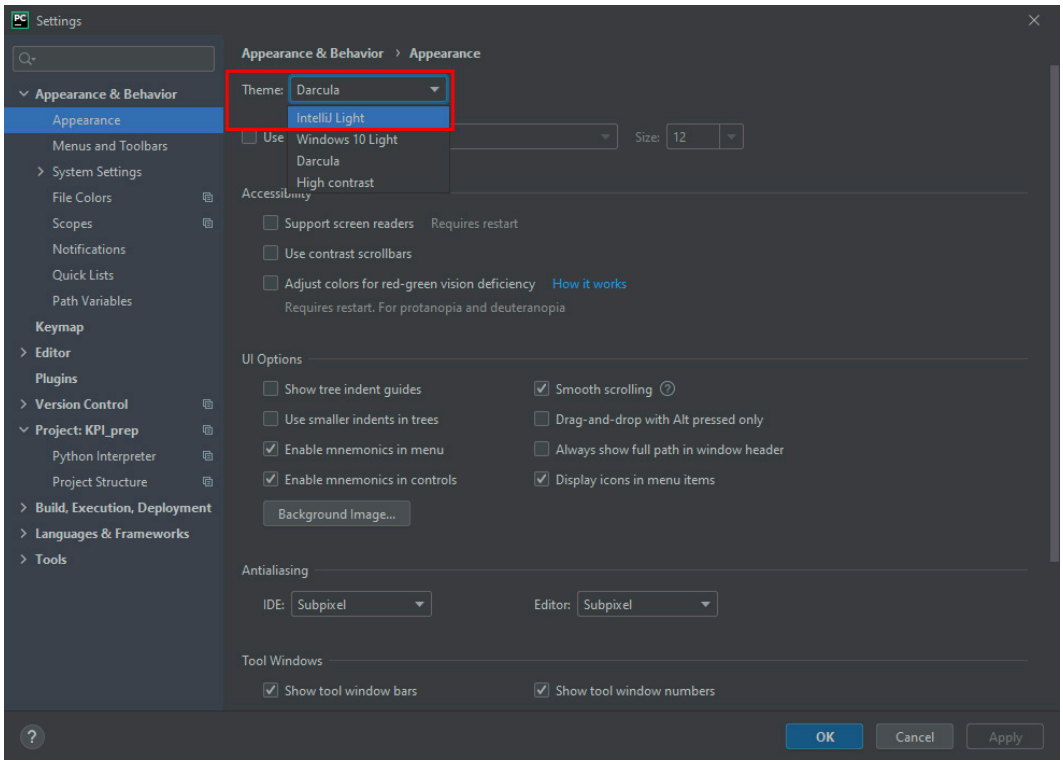


Рис. 1.53. Режим зміни теми

Створення файлу

10.Створюємо файл demo.py. Для цього виконуємо такі дії:

1. Активізуємо рядок з папкою з іменем проекту.

2. Вибираємо в меню File-> New…->Python File.

3. Відкривається вікно для вводу імені файлу:

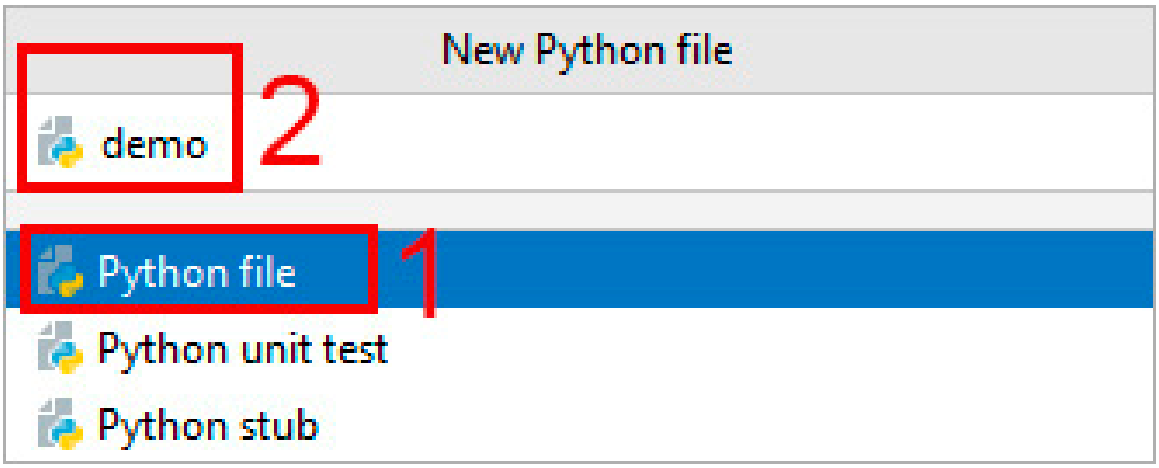


Рис. 1.53. Вікно створення файла

11.У новому файлі, який відкрився у полі редактора коду, запишемо:

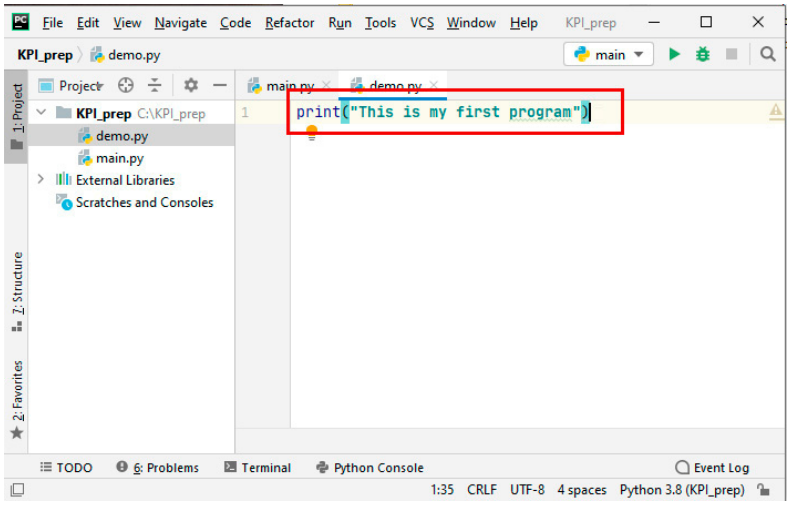


Рис. 1.54 Перша програма

print() – це функція для виводу даних, які записані у круглих дужках

Всі текстові константи беремо в лапки. Отже дана програма виведе текстову константу. В лапках можна записувати довільну послідовність сисмволів.

Запуск програми

12.Запуск програми в PyCharm виконуємо за допомогою комбінації клавіш Alt+Shift+F10 з наступним вибором імені файлу у спливаючому меню. Після цієї операції даний файл можна запускати з використанням комбінації Shift+F10.

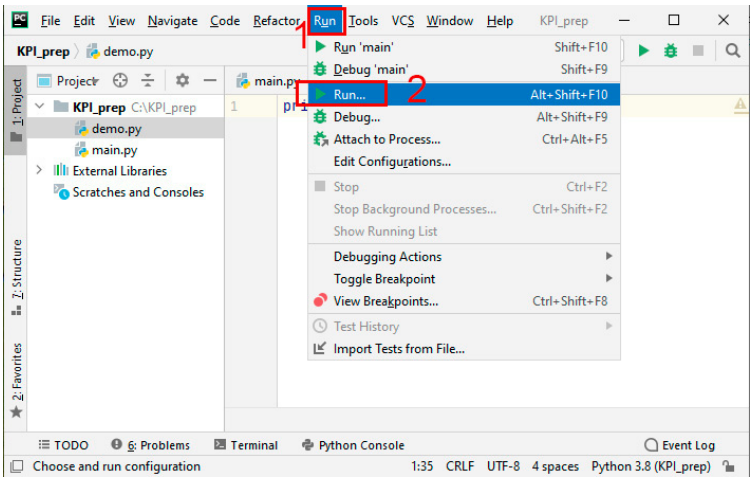


Рис. 1.55. Режим запуску програми

13.Після виконання програми у нижній частині екрана автоматично відкриється вікно «Run», у якому будуть відображені результати роботи або можливі помилки. Скріншот фрагмента вікна «Run» для нашого випадку.

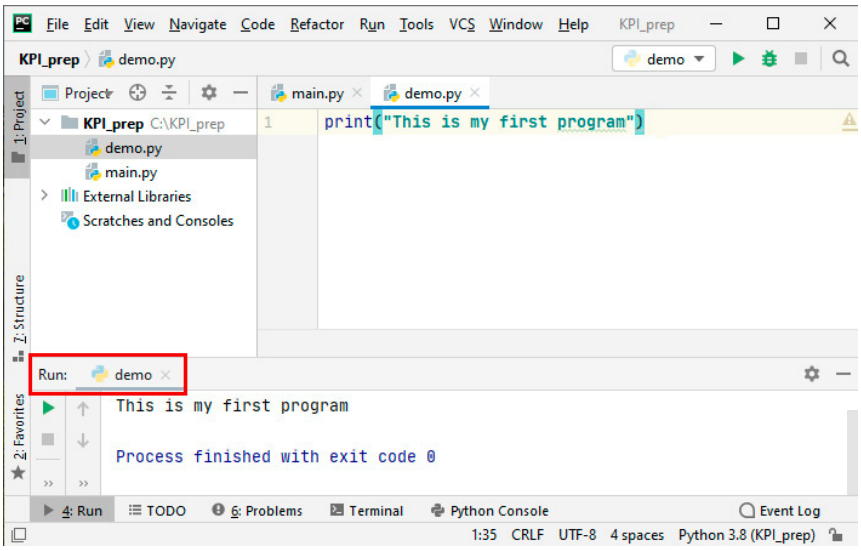


Рис. 1.56. Вікно виводу результатів

Робота з інтерпретатором

14.Для роботи з інтерпретатором треба відкрити вікно «Python Console», як показано на скріншоті

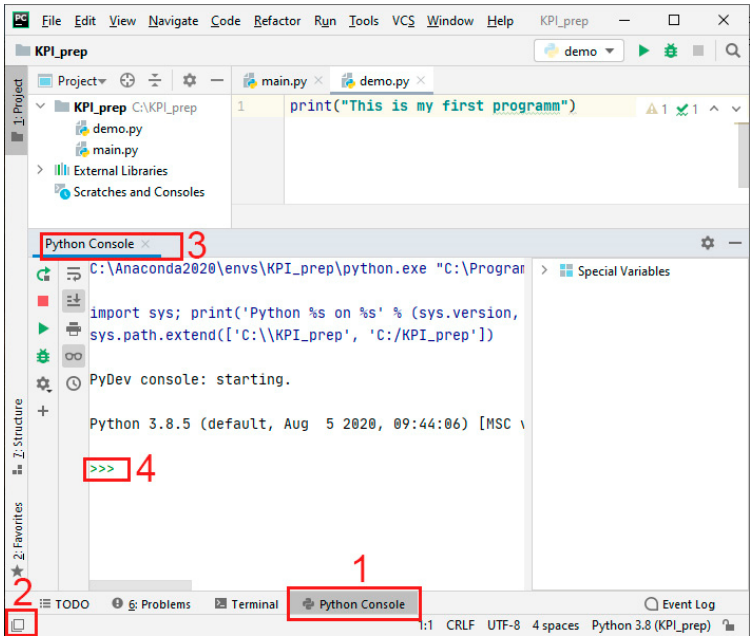


Рис. 1.57. Python Consol

Відкрити вікно можна натиснувши кнопку 1 або відкривши меню 2. На місці вікна «Run» відкривається вікно «Python Console» 3. Про готовність інтерпретатора до роботи свідчить запрошення 4

Виконання команди в інтерпретаторі

15. Введемо команду роздруківки рядка і натиснемо «Enter»

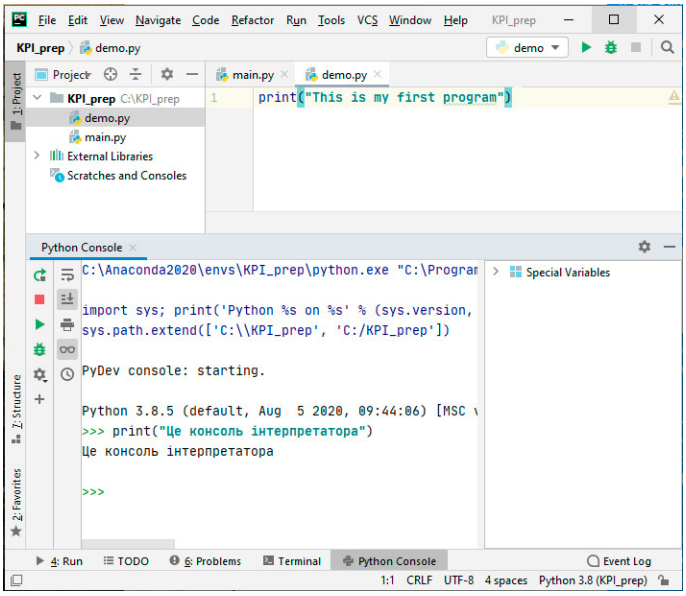


Рис. 1.58. Роздруківка в режимі інтерпретатора

Рядком нижче інтерпретатор виводить результат своєї роботи.

П*рисвоєння значення константи змінній*

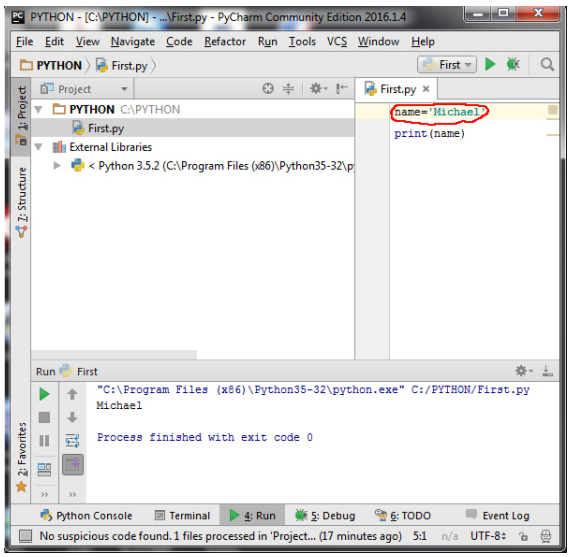


Рис. 1.59. Присвоєння значення змінній

У цьому випадку значення константи ‘Michael’ присвоєне змінній name, а потім виведений вміст змінної name.

2б. Випадок задавання даних шляхом уведення з консолі. Будемо використовувати функцію input.

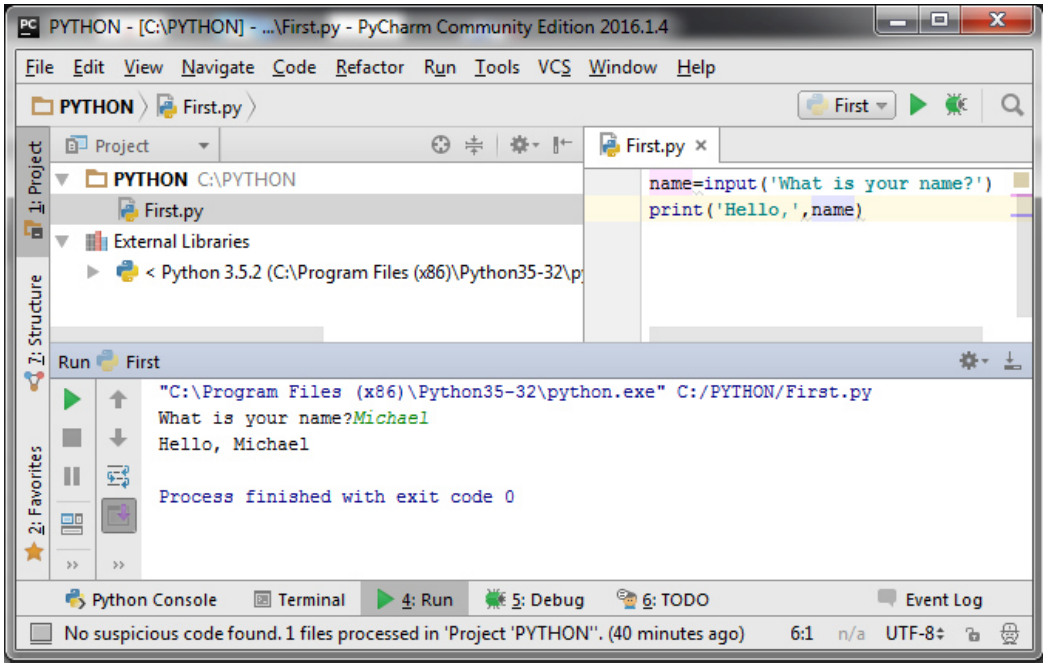


Рис. 1.60. Використання функції input()

Функції input і print – це вбудовані в Python стандартні програми, які забезпечують введення даних з консолі й вивід на консоль.

Функції input і print в інтерактивній програмі

```
name=input('What is your name?')
print('Hello ', name)
```

а) Виводиться на консоль текст:

```
What is your name?
```

б) Програма переходить в очікування введення

в) Вводимо ім'я й натискаємо: `Enter`

г) Виводиться на консоль спочатку константа Hello, а потім – вміст змінної name.

Функція print у цьому випадку виводить два значення через кому: константу ‘Hello,’ і змінну name.

Рядки в програмі на Python

Програма на Python, з «погляду» інтерпретатора, складається з **логічних рядків**.

Один логічний рядок, як правило, розташовується в одному фізичному.

Як записують довгі рядки

1. Явний запис. Логічний рядок може складатися з декількох фізичних. Для переносу використовують символ «\». Наприклад:

```
print (a, " - дуже довгий рядок, який не міститься в", \
80, "знакомісцях")
```

2. Неявний запис. Логічний рядок не закінчується, поки не закрита дужка

Символ-Ґрати (#) `відзначає коментар до кінця рядка`

1.3.5.Запуск Jupyter notebook

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

В результаті відкриваємо вікно Jupyter такого вигляду.

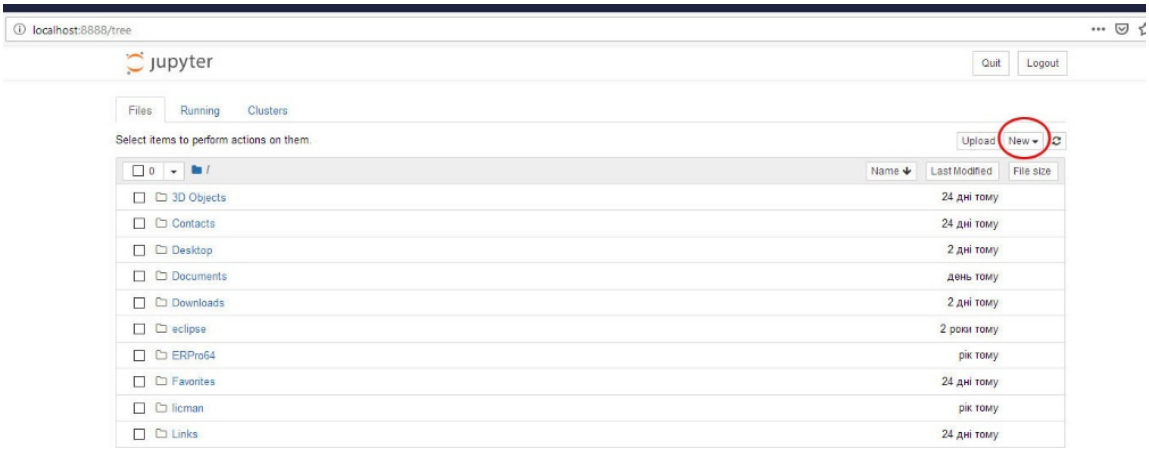


Рис. 1.61. Каталог зошитів Jupyter

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

Вікно інтерпретатора Python в Jupyter notebook має такий вигляд:

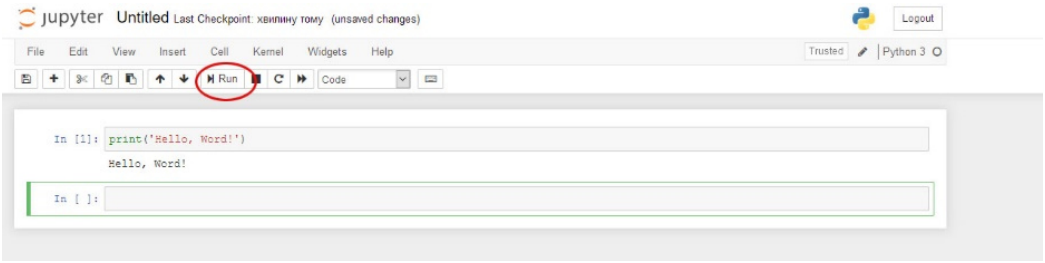


Рис. 1.62. Запуск коду в jupyter

Для виконання команди необхідно натиснути кнопку «Run» на панелі інструментів.

Для створення зошита в Jupyter notebook виконаємо такі дії:

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 1.63. Виклик терміналу Windows

3. У відкритому вікні консолі викликаємо Jupyter notebook.

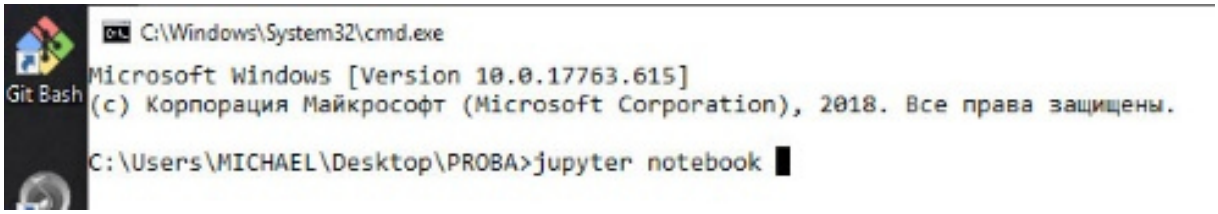


Рис. 1.64. Запуск Jupyter Notebook у терміналі

4. В результаті відкривається пустий зошит Jupyter.

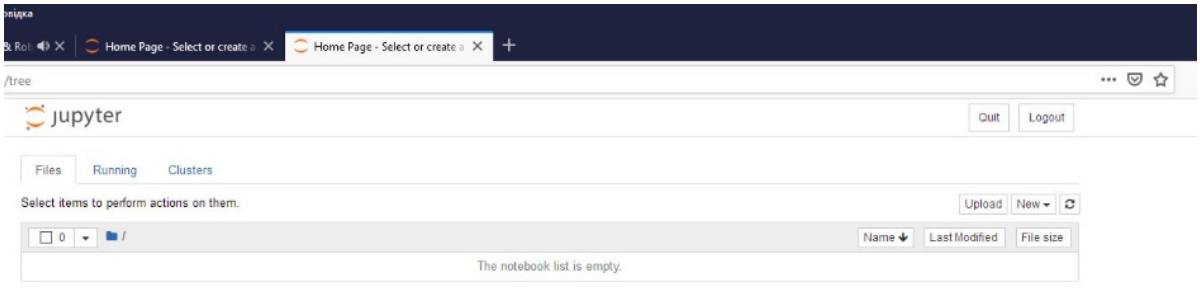


Рис. 1.65. Вікно зошита Jupyter

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

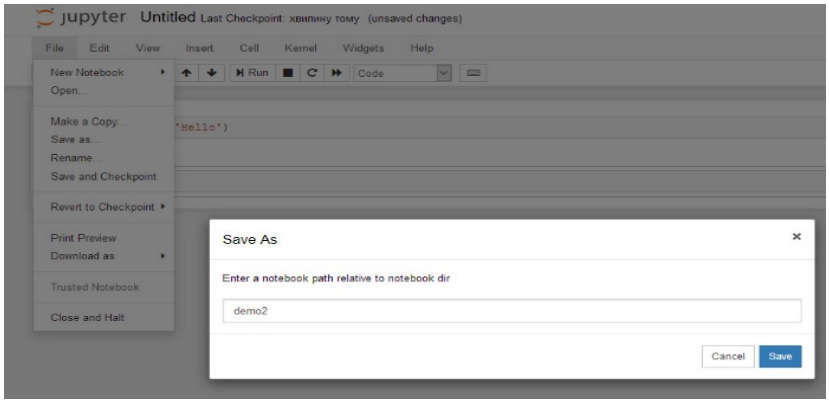


Рис. 1.66. Вікно збереження зошита

6. Файл demo2 буде збережено у відповідній папці

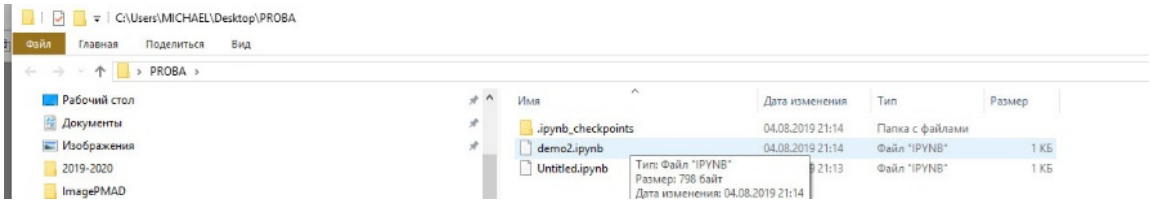


Рис. 1.67. Зберігання файлів у папці Windows

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.

1.3.6. Приклади роботи з Jupyter notebook

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

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

Створення папки

Для створення папки натискаємо на New у правій частині екрану і вибираємо у відповідному меню пункт Folder.

Рис. 1.68. Створення папки в середовищі Jupyter Notebook

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. 1.69. Зміна назви папки

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. 1.70. Вибір типу зошита Jupyter

Створення зошита

Рис. 1.71. Створення комірки

Код мовою програмування Python або текст в нотації Markdown потрібно вводити в комірки:

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

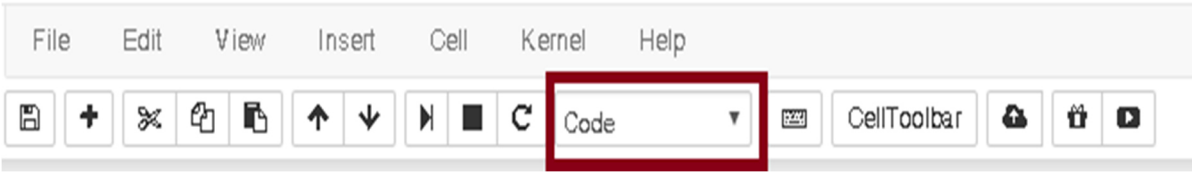


Рис. 1.72. Створення комірки для коду

Якщо вводимо Markdown текст, то выставляємо “Markdown”.

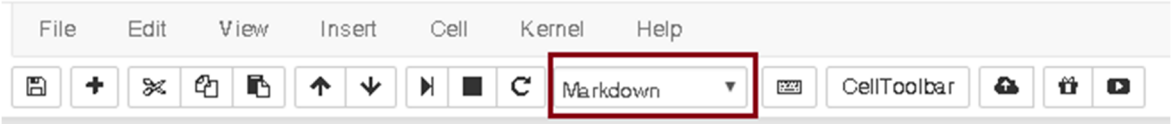


Рис. 1.73. Створення комірки для коментарів

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.

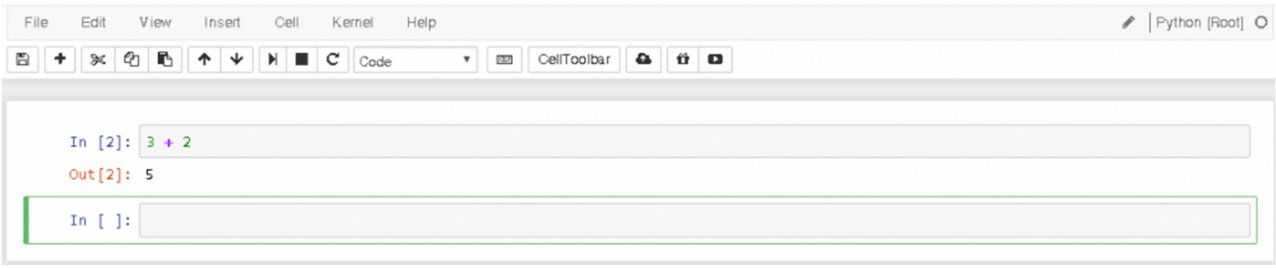


Рис. 1.74. Виконання коду одночасно зі створенням комірки

Якщо у вас вийшло це зробити, виконайте ще кілька прикладів:

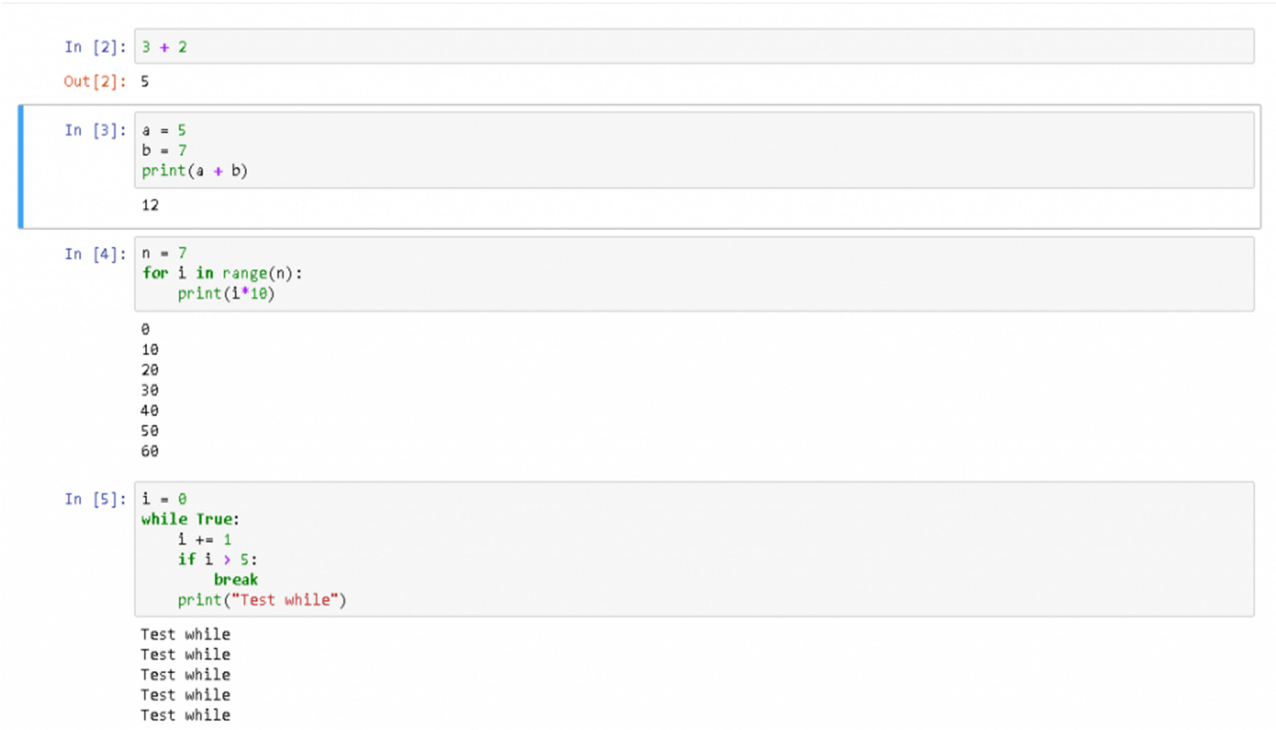


Рис. 1.75. Приклади запуску і виконання коду

Основні елементи інтерфейсу Jupyter notebook

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

Рис. 1.76. Вікно перейменування зошита

Елементи інтерфейсу включають панель меню:

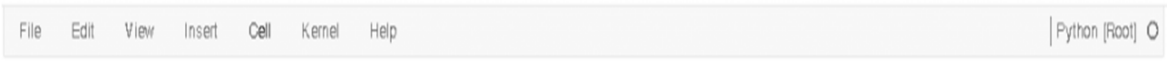


Рис. 1.77. Головне меню

Панель інструментів.

Рис. 1.78. Панель інструментів

Робоче поле з комірками:

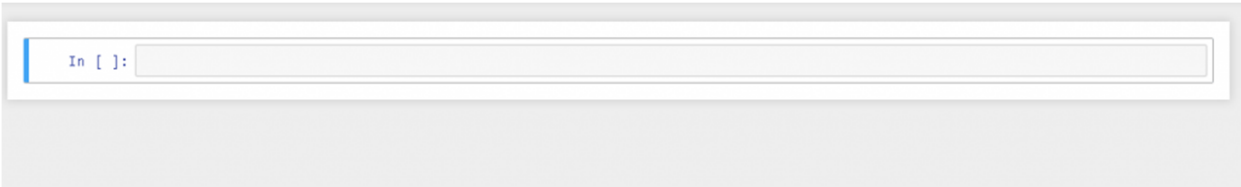


Рис. 1.79. Комірка зошита

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.

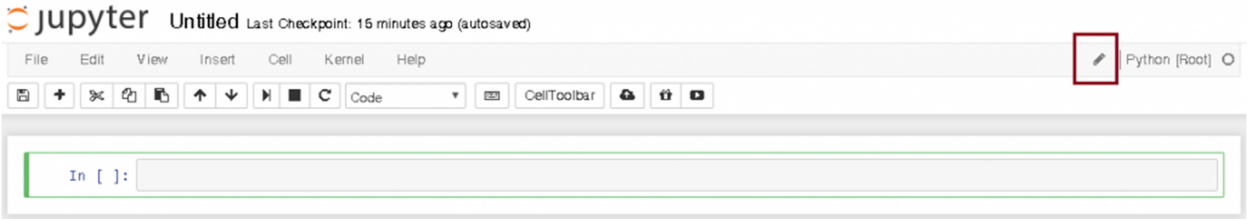


Рис. 1.80. Командний режим роботи

Щоб відкрити текст довідки по сполученнях клавіш, натисніть "Help-> Keyboard Shortcuts"

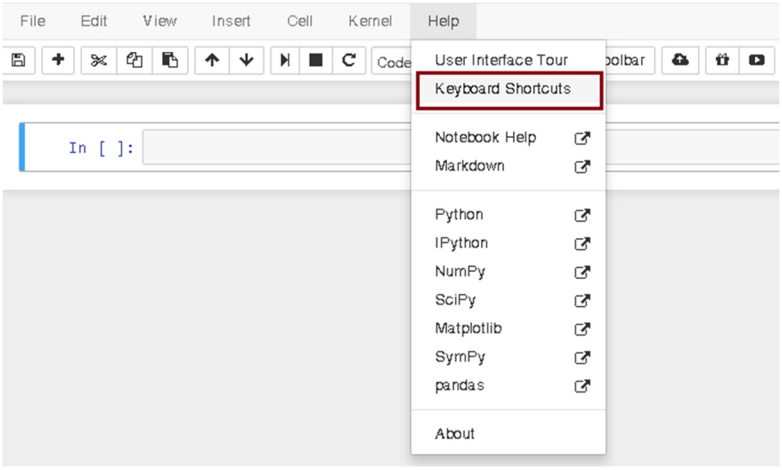


Рис. 1.81. Меню допомоги

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

Рис. 1.82. Режим очікування

Якщо ядро виконує завдання, то зображення зміниться на зафарбований круг.

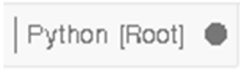


Рис. 183. Режим роботи

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 і 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.

Як зробити ноутбук доступним для інших людей.

Існує кілька способів поділитися своїм ноутбуком з іншими людьми, 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/.

Вивід зображень у ноутбуці

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:

```
from matplotlib import pylab as plt
%matplotlib inline
```

Для виоконання цього коду бібліотека matplotlib повинна бути попередньо завантажена у ваше середовище

Приклад виведення графіка представлений на скріншоті нижче.

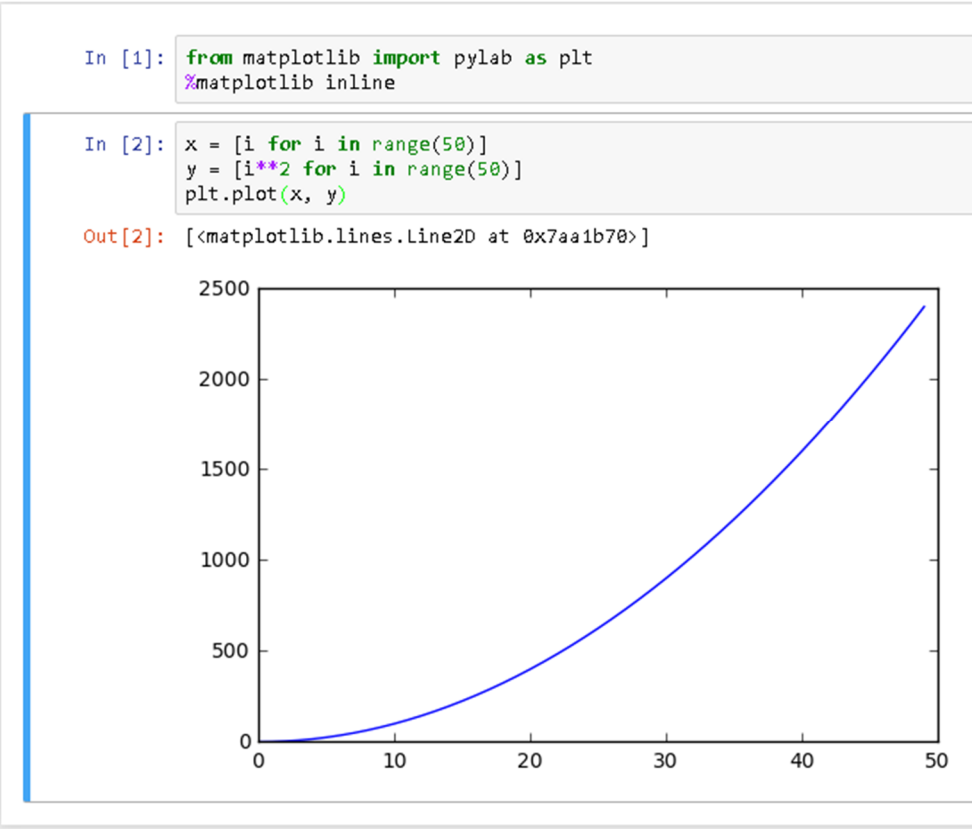


Рис. 1.84. Графік у зошиті Jupyter

Приклад створення власного зошита

1.Запускаємо Jupyter notebook в папці проекту

2.Створюємо новий зошит вибраши в меню New->Python 3

3.Перейменовуємо зошит, клікнувши 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

і натискаємо Shift+Enter

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

(пробіл) Група ІО-11

і натискаємо Shift+Enter

8.У третій комірці пишемо:

print('Hello, World!!')

і натискаємо Ctrl+Enter

9.Отримуємо приблизно такий результат:

10. Зберігаємо свій перший

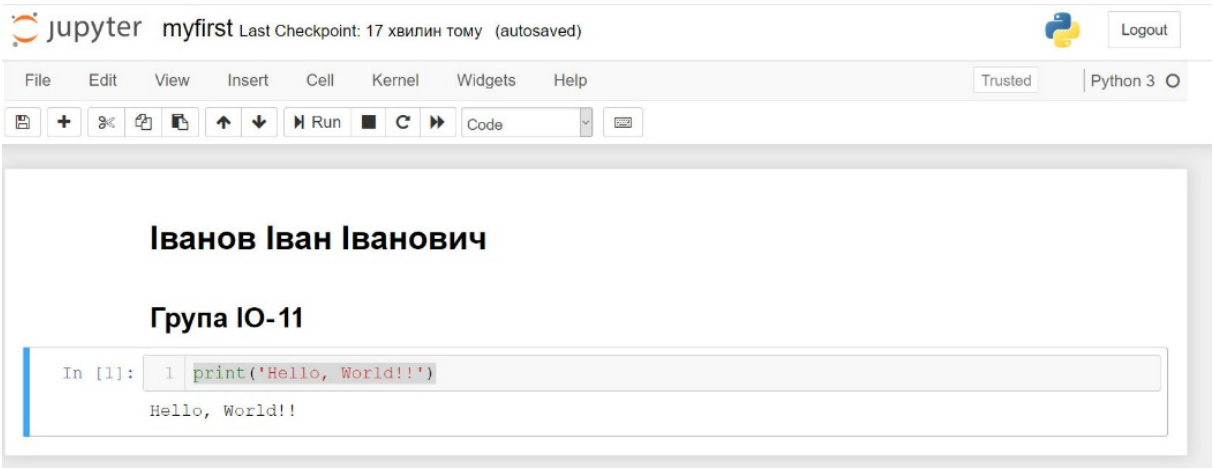


Рис. 1.85. Приклад зошита Jupyter

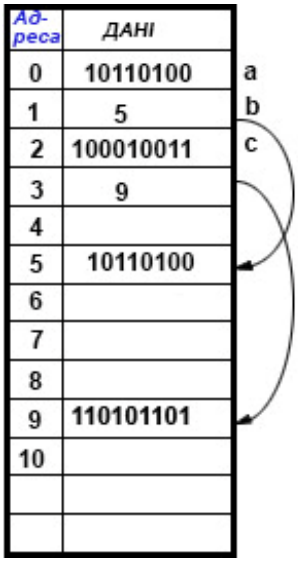
Контрольні запитання до розділу 1.

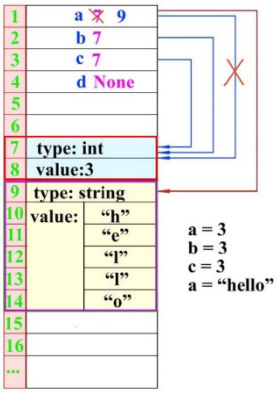
1. Які операції виконує технічне обладнання комп’ютера ?
2. Які типи побутових комп’ютерів вам відомі?
3. На яких принципах базується архітектура комп’ютерів?
4. Дайте визначення комп’ютерної програми.
5. Які елементи програмного забезпечення утворюють програмну конфігурацію?
6. Перерахуйте складові частини программної системи комп’ютера.
7. Перерахуйте переваги мови програмування Python.
8. Які два основні вікна включає програма IDLE?

2. ТИПИ ДАНИХ І ЗМІННІ У PYTHON

2.1. Огляд типі даних у мові Python

Змінна величина в математиці — символ, що позначає будь-яке число в алгебраїчному виразі.





$$a^2 + b^2 = c^2 \begin{cases} 3^2 + 4^2 = 5^2 \\ 5^2 + 12^2 = 13^2 \\ 7^2 + 24^2 = 25^2 \end{cases}$$

Змінна (програмування) — поіменована, або адресована у інший спосіб, область пам'яті, адресу якої можна використовувати для здійснення доступу до даних і змінювати значення в ході виконання програми.

2.1.1. Представлення даних у Python

Усі дані в мові Python представлені об'єктами.

Кожний об'єкт має тип даних і значення.

Для доступу до об'єкта призначені змінні.

При ініціалізації в змінній зберігається посилання на об'єкт (адреса об'єкта в пам'яті комп'ютера).

Завдяки цьому посиланню можна надалі змінювати об'єкт із програми.

Для вибору імені змінної необхідно строго дотримуватись:

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* із 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 1.

Приклад 2.1. 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 2.2. 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.

~~п_125~~

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

my_age = 17, my_height = 175

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

```
>>> x = 10; # Мала буква
>>> X = 20  # Велика буква
>>> x, X
(10, 20)
```

2.1.2. Типи даних

bool - логічний тип даних. (числовий тип)

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

```
>>>logvar=True
>>>type(logvar)
<class 'bool'>
>>> 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 2.3.

```
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 2.4
`>>> 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 2.4а
	`>>> True + True`
	`2`
	`>>> True - False`
	`1`
	`>>> True * False`
	`0`

2.1.3. 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 2.5. 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,1E,1F,20,21,22,23,24,25,26,27,28,29,3A,…
>>> 0xDEC1F
912415
```

2.1.4. 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 2.6.

```
>>> float(5.4)
5.4
>>> float(8.9e-4)
0.00089
>>> float(8.9e-12)
8.9e-12
```

2.1.5. 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 2.7

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

2.1.6. 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 2.8

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')
```

2.1.7. Тип Unicode-Рядки str (Послідовність)

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

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

Приклад 2.9.

```
>>> 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

2.1.8. Типи bytes і 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)

2.1.9. 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 2.10.

```
>>> 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']
```

2.1.10. Тип tuple — кортеж (Послідовність)

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

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

Приклад 2.11.

```
>>> 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)
```

2.1.11. Тип range — діапазон (Послідовність)

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

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

Приклад 2.12

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

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

Приклад 2.13

```
>>>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 2.14

```
>>>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 ми використовуємо негативний крок

Приклад 2.15

```
>>>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
```

2.1.12. Тип dict — словник

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 2.16

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

2.1.13. 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 2.17

```
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'}
```

2.1.14. Тип function — функція:

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

Тип **module** — модуль:

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

Тип **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'>
```

2.1.15. Змінювані й незмінювані типи

Змінювані типи: списки, словники й тип bytearray.

Приклад15. 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 2.18.

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

2.1.16. 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 2.19. 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 2.20.

```
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 2.21. Вивід елемента списку:

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

Приклад 2.22. Перебрати слово по буквах:

```
for i in "Рядок": print(i)
Р
я
д
о
к
```

Приклад 2.23. Перебір елементів словника:

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

1
2
```

2.2. Основи роботи зі змінними мові Python

2.2.1. 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
>>> x = 0xAF       #Тип int
>>> y = 7.8        #Тип float
>>> s1 = 'Рядок' # Тип str
>>> b = True       # Тип bool
```

В одному рядку можна присвоїти значення відразу декільком змінним:

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

>>> x = y = z = (1,2,3,4,5)
>>> print(x)
(1,2,3,4,5)
>>> print(y)
(1,2,3,4,5)
>>> print(z)
(1,2,3,4,5)
```

2.2.2. 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 2.24. Пояснення, 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 кортеж 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])
```

2.2.3. 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 2.25.

```
>>> 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
```

2.2.4. Об'єднання посилань при кешируванні

З метою підвищення ефективності коду інтерпретатор виконує кеширування 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 2.26.

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

2.2.5. Перевірка кількості посилань

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.

2.2.6. Позиційне присвоювання

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

Приклад 2.28. Позиційне присвоювання:

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

За допомогою позиційного присвоювання можна поміняти значення змінних місцями.

Приклад 2.29.

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

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

По обидві сторони оператора = можуть бути зазначені послідовності.

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

Приклад 2.30.

```
>>> 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)
```

2.2.7. Питання відповідності елементів при обміні

Зверніть увагу на те, що кількість елементів праворуч і ліворуч від оператора = повинна збігатися, інакше буде виведене повідомлення про помилку:

```
>>> x, y, z = (1, 2, 3, 4)
Traceback (most recent call last):
File "<pyshell#130>", line 1, in <module>
x, y, z = (1, 2, 3, 4)
ValueError: too many values to unpack (expected 3)
```

Python 3 при невідповідності кількості елементів праворуч і ліворуч від оператора = дозволяє зберегти в змінній список, що складається із зайвих елементів. Для цього перед іменем змінної вказується зірочка (*).

Приклад 2.31.

```
>>>x,y,*z=(1,2,3,4)
>>>x,y,z
(1,2,[3, 4])
>>>x,*y,z = (1,2,3,4)
>>>x,y,z
(1,[2,3],4)
>>>*x,y,z =(1,2,3,4)
>>>x,y,z
([1,2],3,4)
>>>x,y,*z =(1,2,3)
>>>x,y,z
(1,2,[3])
>>>x,y,*z=(1,2)
>>>x,y,z
(1,2,[])
```

Як видно із прикладу, змінна, перед якою зазначена зірочка, завжди містить список. Якщо цій змінній не вистачило значень, то їй присвоюється порожній список.

Слід пам'ятати, що зірочку можна вказати тільки перед однією змінною.

В протилежному випадку виникне неоднозначність і інтерпретатор виведе повідомлення про помилку:

```
>>> *x, y, *z = (1, 2, 3, 4)
File "<input>", line 1
SyntaxError: two starred expressions in assignment
```

2.2.8. Перевірка типу даних

Python у будь-який момент часу змінює тип змінної відповідно до даних, що зберігаються в ній.

Приклад 2.32.

```
>>> a = 'Рядок'    #тип str
>>> a = 7          #тепер змінна має тип int
```

Визначити, на який тип даних посилається змінна, дозволяє функція `type (<змінної>):`

```
>>> type(a)
<class 'int'>
```

Перевірити тип даних, що зберігаються в змінній, можна такими способами:

- порівняти значення, що повертається функцією **type()**, з назвою типу даних:

```
>>> x = 10
>>> if type(x) is int: print("Це тип int")
```

- перевірити тип за допомогою функції **isinstance():**

```
>>> s = "Рядок"
>>> if isinstance(s, str):
    print("Це тип str")
```

Це тип str

2.2.9. Принципи формування типів даних

1. Після присвоювання значення в змінній зберігається посилання на об'єкт певного типу, а не сам об'єкт.
2. Якщо потім змінній присвоїти значення іншого типу, то змінна буде посилатися на інший об'єкт, і тип даних відповідно зміниться.
3. Таким чином, тип даних у мові Python — це характеристика об'єкта, а не змінної. Змінна завжди містить тільки посилання на об'єкт.

Операції залежать від типу значення

Після присвоювання змінній значення над цим значенням можна виконувати операції, призначені лише для даного типу даних. Наприклад, рядок не можна скласти з числом, тому що це приведе до виводу повідомлення про помилку:

```
>>> 2 + "25"
Traceback (most recent call last):
File "<pyshell#o>", line 1, in <module>
+ "25"
TypeError:unsupported operand type(s) for+: 'int
'and'str'
```

2.2.10. Функції для перетворення типів даних

1. Перетворення об'єкта в логічний тип даних

```
bool(<об'єкт>)
```

Приклад 2.33.

```
>>> bool(""), bool("Рядок")
(False, True)
>>> bool(0), bool(10)
(False, True)
>>>bool([]),bool([1,2])
(False,True)
```

2. Перетворення об'єкта в число

```
int(<об'єкт>[,<Система числення>])
```

У другому параметрі можна вказати систему числення (значення за замовчуванням — 10).

Приклад 2.34

```
>>> int(7.5),int("71")
```

```
(7, 71)
>>>int("71",10),
71
>>>int("71",8), int("0o71",8)
(57, 57)
>>>int("A",16)
10
```

3. Перетворення цілого числа або рядка в дійсне число.

float([<Число рядок>])

Приклад 1.35.

```
>>> float(7),float("7.1")
(7.0, 7.1)
>>> float("Infinity"), float("-inf")
(inf, -inf)
>>> float("Infinity") + float("-inf")
nan
```

4. Перетворення об'єкта в рядок

str([<Об'єкт>])

Приклад 2.36:

```
>>> str(125), str([1, 2, 3])
('125', '[1,2,3]')
>>> str((1,2,3)), str({"x": 5, "в": 10})
('(1, 2, 3)', "{'x': 5, 'y': 10}")
>>> str( bytes("м", "cp1251") )
b'\xec'
>>> str( bytes("f", "cp1251") )
"b'f'"
>>> str( bytearray("З", "cp1251"))
```

```
"bytearray(b'\\xc7')"
```

5. Перетворення послідовності в список

list(<послідовність або множина або словник>)

Приклад 2.37:

```
>>> list("12345")       #Рядок str⟹ list
['1', '2', '3', '4', '5']
>>> list("Рядок")       #Рядок  str⟹ list
['Р', 'я', 'д', 'о', 'к']
>>> list({1,2,3,4,5})  #Множина set⟹ list
[1, 2, 3, 4, 5]
>>> list({"x": 5, "y": 10})
 ['x', 'y']             #Словник  dict⟹ list
```

6. Перетворення послідовності в кортеж

tuple((<послідовність або множина або словник>)

Приклад 2.38

```
 >>> tuple("123456") #Рядок str⟹ tuple
('1', '2', '3', '4', '5', '6')
>>> tuple([1,2,3,4,5]) #Список list⟹ tuple
(1, 2, 3, 4, 5)
>>> tuple(({"x": 5, "y": 10}))
('x', 'y')             #Словник dict⟹ tuple
```

Приклад з користувацьким введенням даних.

Розглянемо можливість додавання двох чисел, введених користувачем.

Вводити дані дозволяє функція **input().**

Приклад 2.39.

```
>>>x = input("x = ")    # Вводимо 5
```

```
>>>y  =  input("y  =  ")      #
Вводимо 12
>>>print (x + y)
512
>>>x ="str"
>>>y ="ing"
>>>print(x+y)
string
```

Результатом виконання цього скрипта є не число, а рядок 512. Таким чином, слід запам'ятати,
що функція input() повертає результат у вигляді рядка.
Щоб додати два числа, необхідно перетворити рядок на число

Приклад 2.40. Перетворення рядка в число

```
>>>x = int(input("x = ")) # Вводимо 5
>>>y = int(input("y = ")) # Вводимо 12
>>>print (x + y)
17
```

У цьому випадку ми одержимо число 17, як і повинно бути. Однак якщо користувач замість числа введе рядок, то програма завершиться з фатальною помилкою.

```
x = "w"
Traceback (most recent call last):
  File "C:/PYTHON/Samples.py", line 1, in <module>
    x = int(input("x = "))   # Уводимо 5
ValueError: invalid literal for int() with base 10: 'w'
```

2.2.11. Видалення змінної

Вилучити змінну можна за допомогою інструкції del:

```
del <Змінна1>[, ... , <Змінна n>]
```

Приклад 2.41. Видалення однієї змінної:

```
>>> x=20
```

```
>>>x
20
>>> del x
>>> x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'x' is not defined
```

Приклад видалення декількох змінних:

```
>>> x, y = 10, 20
>>> del x, y
```

2.2.12. Функція `print()`

Функція **print**() виводить задані в параметрах об'єкти на екран або на інший стандартний пристрій.

Довільний об'єкт буде перетворено в рядок перед виводом.

```
print(object(s), sep='separator', end='end')
```

Приклад 2.42.

```
>>>print(1,2,3,4)
1 2 3 4
>>>print(1,2,3,4,sep=",")
1,2,3,4

>>>for i in [1,2,3,4]: print(i, sep=',')
1
2
3
4

>>>for i in [1,2,3,4]: print(i, end=',')
1,2,3,4,
```

Контрольні запитання до розділу 2

1. Які іменна змінних задані правильно: `b1, ф2, 1n, wen_1`?
2. Наведіть числові типи даних у мові Python.
3. Який спеціальний тип даних використовується у мові Python?
4. Перерахуйте типи даних, які представляють послідовності.
5. Чим відрізняється групове присвоювання від позиційного просвоювання значень змінним?
6. В чому полягає принцип кеширування зміних у мові Python?
7. Як можливо перевірити тип даних змінних?
8. За якими правилами формують назви функцій для перетворення типів даних?

3. ОПЕРАТОРИ У МОВІ PYTHON

Оператор у мові Python – це певний визначений символ, який задає фіксоване правило виконання операцій над одним або кількома операндами.

Операнди – це змінні або літерали, над якими виконується операція, що задана оператором.

Приклад 3.1.

```
a,b=10,20 # Задаємо значення операндам
result=a+b # Операція додавання над операндами
print (a, "+",b,"=",result) # Вивід результату
```

Результат виконання:

10 + 20 = 30

Типи операторів

У мові Python визначено 7 типів операторів:

1. Арифметичні оператори
2. Побітові оператори
3. Оператори для послідовностей
4. Оператори присвоювання
5. Оператори порівняння
6. Логічні оператори
7. Оператори тотожності

Розглянемо оператори, доступні в Python 3, докладніше.

3.1. Арифметичні оператори

3.1.1. Додавання

Цей оператор додає значення, які розташовані по обидві сторони оператора.

+ — додавання: $12+3=15, 10.4+5.4=15.8, 10+11.2=21.2$

```
# Цілі числа
>>> 1 2 + 3                    print(12+3)
15                             Результат: 15
#Дійсні числа
>>> 10.4 + 5.4                 print(10.4+5.4)
15.8                           Результат: 15.8
#Цілі й дійсні числа
>>> 10 + 11.2                  print(10+11.2)
21.2                           Результат: 21.2
```

3.1.2. Віднімання

Цей оператор віднімає значення, яке розташоване справа від оператора від значення, яке розташоване зліва від оператора.

– — віднімання: $11-3=8,\ 14.4-5.2=9.2,15-6.2=8.8$

```
# Цілі числа
>>> 1 2 - 3                    print(11-3)
8                              Результат: 8
#Дійсні числа
>>> 14.4 - 5.2                 print(14.4-5.2)
9.2                            Результат: 9.2
#Цілі й дійсні числа
>>> 15 - 6.2                   print(15-6.2)
8.8                            Результат: 8.8
```

3.1.3. Множення

Цей оператор перемножує значення, які розташовані по обидві сторони оператора.

***** — множення: $60\times5=300,\ 12.4\times5.2=64.48,10\times5.2=52.0$

```
# Цілі числа
>>> 60 * 5                  print(60 * 5)
300                         Результат: 300
#Дійсні числа
>>> 12.4 * 5.2              print(12.4 * 5.2)
64.48                       Результат: 64.48
#Цілі й дійсні числа
>>> 10 * 5.2                print(10*5.2)
52.0                        Результат: 52.0
```

3.1.4. Ділення

Цей оператор ділить значення, яке розташоване зліва від оператора на значення, яке розташоване справа від оператора.

/ — ділення**.** $10/5=2, 10/3=3.333...$

Особливість. Результатом ділення завжди є дійсне число, навіть якщо проводиться ділення цілих чисел.

```
# Ділення цілих чисел без остачі
>>> 10/5                    print(10/5)
2.0                         Результат: 2.0
# Ділення цілих чисел з остачею
>>> 10/3                    print(10/3)
3.3333333333333335          Результат:
                            3.3333333333333335
# Ділення дійсних чисел
>>> 10.2/5.7                print(10.2/5.7)
1.789473684210526           Результат:
                            1.789473684210526
```

3.1.5. Ділення з округлення до меншого

/ / — Ділення з округленням до меншого числа. Незалежно від типу чисел остача відкидається: $\lfloor 10/5 \rfloor = 2, \lfloor 10/3 \rfloor = 3$

Приклад 3.2.

```
>>> 10 // 5 # Ділення цілих чисел без остачі
2
>>> 10 // 3  # Ділення цілих чисел з остачею
3
>>> -10 // 3  # Ділення цілих чисел з остачею
-4
>>> 10.0 // 5.0 # Ділення дійсних чисел
2.0
>>> 10.0 // 3.0  # Ділення дійсних чисел
3.0
>>> -10 // 5.0 # Ділення цілого числа на дійсне
-2.0
>>> 10 // 3.0 # Ділення цілого числа на дійсне
3.0
```

3.1.6. Залишок від ділення

% — залишок від ділення: $10 \bmod 5 = 0, 10 \bmod 3 = 1$

Приклад 3.3.

```
>>> 10 % 5  #Залишок від ділення цілих чисел без
остачі
0
>>> 10 % 3  # Залишок від ділення цілих чисел з
остачею
1
>>> 10.0 % 5.0 # Операція над дійсними числами
0.0
>>> 10.0 % 3.0 # Операція над дійсними числами
1.0
>>> 10%5.0 # Операція над цілими й дійсними числами
0.0
>>> 10%3.0 # Операція над цілими й дійсними числами 1.0
```

3.1.7. Піднесення до степеню

Цей оператор підносить значення, яке розташоване зліва від оператора до степеню, який розташований справа від оператора.

****** — піднесення до степеня: $10^2 = 100,\ 10.0^2 = 100.0$

```
>>> 10 ** 2       #  Результат int
100
>>> 10.0 ** 2     # Результат float
100.0
>>> 10 ** 2.0     # Результат float
100.0
>>> 10.0 ** 2.0   # Результат float
100.0
>>> 10.3 ** 2.123
141.33620568286585
```

3.1.8. Унарна операція зміни знаку

Унарний плюс (+) :

>>>a=10 >>>+a >>>10 #+(+10)=10	>>>a=-10 >>>+a >>>-10 #+(-10)=-10
>>>a=10.0 >>>+a >>>+10.0#+(+10.0)=10.0	>>>a=-10.0 >>>+a #+(-10.0)=-10.0 >>>-10.0

Унарний мінус (-) :

>>>a=10 >>>-a >>>-10 #-(+10)=-10	>>>a=-10 >>>-a >>>10 #-(-10)=10
>>>a=10.0 >>>-a >>>-10.0#-(+10.0)=-10.0	>>>a=-10.0 >>>-a #-(-10.0)=10.0 >>>10.0

3.1.9. Висновок до застосування арифметичних операторів

Операції над числами різних типів повертають число, що має більш складний тип з тих типів, що брав участь в операції.

Складність типів розглядають в такому порядку:

1. Цілі числа мають найпростіший тип
2. Наступними по складності йдуть дійсні числа
3. Найскладніший тип — комплексні числа.

Таким чином, якщо в операції беруть участь ціле число й дійсне, то ціле число буде автоматично перетворене в дійсне число, а потім виконана операція над дійсними числами.

Результатом цієї операції стане дійсне число.

3.1.10. Точність представлення чисел в Python

При виконанні операцій над дійсними числами слід враховувати обмеження точності обчислень. Наприклад, результат наступної операції може здатися дивним:

Приклад 3.4.

```
>>> 0.3-0.1-0.1-0.1
-2.7755575615628914e-17
```

Причина. Представлення дробового числа за стандартом IEEE 754 (подвійна точність)

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

де N - число, M - мантиса, n - основа системи числення, p - порядок.

Нехай $N = 12.34$. Тоді $M = 1234$, $n = 10$, $p = -2$

Точне представлення числа можна отримати за допомогою модуля decimal.

```
>>>from decimal import Decimal
>>>Decimal.from_float(0.1)
```

Decimal('0.1000000000000000055511151231257827021181583404541015625')

Якщо необхідно виконувати операції з фіксованою точністю, то слід

використовувати модуль decimal:

Приклад 3.5.

```
from decimal import Decimal
num3= Decimal(“0.3”)
num1= Decimal(“0.1”)
num=num3-num1-num1-num1
print(num)
Результат: 0.0
```

3.2. Побітові оператори

3.2.1. Представлення чисел у двійковій системі числення

Двійкова система числення дозволяє представити довільне число у вигляді числа, що складається тільки з нулів та одиниць. Як і десяткова система числення, двійкова система є позиційною – кожен розряд у числі має свою вагу:

Число у десятковій системі числення

$$a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + a_{n-3}10^{n-3} + ... + a_2 10^2 + a_1 10^1 + a_0 10^0$$

Число у двійковій системі числення:

$$b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + b_{n-3}2^{n-3} + ... + b_2 2^2 + b_1 2^1 + b_0 2^0,$$

де цифри з множини $\{1,0\}$, n-кількість цифр у числі

0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

3.2.2. Побітові оператори

Побітові оператори призначені для маніпуляції окремими бітами. Мова Python підтримує наступні побітові оператори:

~ — двійкова інверсія

Побітова інверсія числа x визначається як $-(x+1)$

Приклад 3.6.

```
>>> x = 0b10      #2
>>> x = ~x
>>> bin(x)        #-(2+1)=-3
'-0b11'
>>> x = -0b11     #-3
>>> x = ~x
>>> bin(x)        # -(-3+1)=2
'0b10'
```

«Амперсанд» & — двійкове AND: (Кон'юнкція)

Таблиця істинності для операції &

0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

Приклад 3.7.

```
>>> x = 0b101
>>> y = 0b110
>>> z = x & y
>>> bin(z)
'0b100'
>>> "{0:b} & {1:b} = {2:b}". format(x, y, z)
'101 & 110 = 100'
```

&	1	0	1
1	1	1	0
	1	0	0

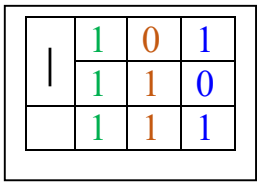
| — двійкове OR: (Диз'юнкція)

Таблиця істинності для операції |

0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

Приклад 3.8.

```
>>> x = 0b101
>>> y = 0b110
>>> z = x | y
>>> bin(z)
'0b111'
>>> "{0:b}|{1:b}={2:b}".format(x, y, z)
'101 | 110 = 111'
```



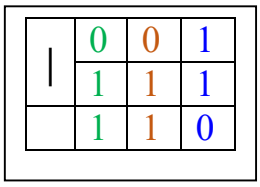
^ — операція XOR: (**Виняткова диз'юнкція**)

Таблиця істинності для операції

0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 0

Приклад 3.9.

```
>>> x = 0b0101
>>> y = 0b1100
>>> z = x ^ y
>>> bin(z)
'0b1001'
>>> "{0:b}^{1:b}={2:b}".format(x, y, z)
'0101 ^ 1100 = 1001'
```



x<<y — зсув вліво – зсуває двійкове представлення числа x вліво на y розрядів, а розряди праворуч заповнюються нулями. Це те саме, що виконати операцію x*(2**y)

```
    1111
   11110 ←
  111100 ←
 1111000 ←
11110000
```

Приклад 3.10.

```
x = 0b1111
print(x, bin(x))
y = x << 1
print(y, bin(y))
z = y << 1
print(z, bin(z))
```

Результат

15 0b1111
30 0b11110
60 0b111100

x>>y - зсув вправо додатного числа –

зсуває двійкове представлення числа x вправо на y розрядів. Це те саме, що виконати операцію x//(2** y)

Приклад 3.11.

Додатне число	**Від'ємне число**
>>> x = 0b11	>>> x=-0b11
>>>y=x>>1; bin(y)	>>> y=x>>1; bin(y)
'0b1'	'-0b10'
Оскільки 3//2=1	Оскільки -3//2=-2
>>>x=0b1001	>>> x=-0b1001
>>>y=x>>1; bin(y)	>>> y=x>>1; bin(y)
'0b100'	'-0b101'
Оскільки 9//2=4	Оскільки -9//2=-5

3.3. Спеціалізовані оператори

3.3.1. Оператори для послідовностей

Для роботи з послідовностями призначені наступні оператори:

+ - конкатенація:

```
>>> print("Рядок1"+"Рядок2") # Конкатенація рядків
Рядок1Рядок2
>>> [ 1, 2, 3] + [ 4, 5, 6]    # Списки
[1, 2, 3, 4, 5, 6]
>>> ( 1, 2, 3) + ( 4, 5, 6)    # Кортежі
(1, 2, 3, 4, 5, 6)
```

Для конкатенації змінних типу dict

```
>>> x={'a':1,'b':2}
>>> y={'c':3,'d':4}
>>> x.update(y)
>>> x
{'d': 4, 'c': 3, 'b': 2, 'a': 1}
```

* - повторення:

Повторити 20 раз символ s

```
>>> "s"*20   # Рядок
'ssssssssssssssssssss'
```

Повторити 3 раза список (list):

```
>>> [1, 2] * 3       # Список
[ 1, 2, 1, 2, 1, 2]
```

Повторити 3 раза кортеж (tuple):

```
>>> ( 1, 2) * 3     # Кортеж
(1, 2, 1, 2, 1, 2)
```

3.3.2. Оператори входження.

in - перевірка на входження.

Якщо елемент входить у послідовність, то повертається логічне значення True:

```
>>> "Рядок" in "Рядок для пошуку" # Рядки
True
>>> 2 in [1,2,3], 4 in [1,2,3]  # Рядки
(True, False)
>>> 2 in (1,2,3), 6 in (1, 2, 3) # Кортежі
(True, False)
>>> 2 in {1,2,3}, 6 in {1, 2, 3} # Множини
(True, False)
>>>  in {1,2,3}, 6 in {1, 2, 3} # Множини
(True, False)
>>>"b" in {"a":123,"b":456,"c":87}
True
```

not in - перевірка на невходження. Якщо елемент не входить у послідовність, повертається True:

Приклад 3.12.

```
>>> "Рядок" not in "Рядок для пошуку" # Рядки
False
>>>"b" not in {"a":123,"b":456,"c":87}
False
>>> 2 not in [1,2,3], 4 not in [1,2,3]   #список
(False, True)
>>> 2 not in (1,2,3), 6 not in (1,2,3)#кортеж
(False, True)
>>> 2 not in {1,2,3}, 6 not in {1,2,3}#множина
(False, True)
```

3.3.3. Оператори присвоювання

Оператори присвоювання призначені для збереження значення в змінній.

= - присвоює змінній значення:

```
>>>x = 5
>>>x
5
```

+= - збільшує значення змінної на зазначену величину:

```
x=y=5
print(x is y)          # True
x+=10
print("x=",x,",y=",y) # x= 15 ,y= 5
print(x is y)          # False
```

Для послідовностей оператор += виконує конкатенацію:

```
>>>m=s="Ряд"
>>>s += "ок"
>>>print(s)
```

```
>>>m is s
False
```

Рядок

-= - зменшує значення змінної на зазначену величину:

```
>>>x = 10
>>>x -= 5 # Еквівалентно x = x - 5
>>>x
5
```

***= -** множить значення змінної на зазначену величину:

```
>>>x = 10
>>>x *= 5 # Еквівалентно x = x * 5
>>>x
50
```

Для послідовностей оператор *= виконує повторення:

```
>>>s = "*"
>>>s*= 20
>>>s
'********************'
```

/= - ділить значення змінної на зазначену величину:

```
>>>x = 10
>>>x /= 3  # Еквівалентно x = x / 3
>>>x
3.3333333333333335
```

//= - ділення з округленням вниз і присвоюванням:

Приклад 3.13.

```
>>>x = 10
>>>x //= 3 # Еквівалентно x = x // 3
>>>x
3
>>>x = -10
>>>x //= 3 # Еквівалентно x = x // 3
>>> x
-4
```

% = - ділення по модулю й присвоювання:

Приклад 3.14.

```
>>>m=x=10
>>>m is x
>>>True
>>>x %= 2  # Еквівалентно x = x % 2
>>>x
0
>>>m is x
>>>False
```

**** =** - піднесення до степеня і присвоювання:

```
>>>x = 10
>>>x**= 2 #Еквівалентно x = x ** 2
>>>x
100
```

3.3.4. Оператори порівняння

== - дорівнює:

```
>>> 1 == 1, 1 == 5
(True, False)
```

!= - не дорівнює:

```
>>> 1 != 5, 1 != 1
(True, False)
```

< - менше:

```
>>> 1 < 5, 1 < 0
(True, False)
```

> - більше:

```
>>> 1 > 0, 1 > 5
(True, False)
>>> a,b=10,20
>>> a==b
False
# Порівняння літералів
```

<= - менше або дорівнює:

```
>>> 1 <= 5, 1 <= 0, 1 <= 1
(True, False, True)
```

>= - більше або дорівнює:

```
>>> 1 >= 0, 1 >= 5, 1 >= 1
(True, False, True)
```

```
# Порівняння змінних
>>> x,y=3,5
>>> x<=y
True
>>> x>=y
False
```

Використання **not** при порівнянні. Значення логічного виразу можна інвертувати за допомогою оператора not:

```
>>>x,y = 1,1
>>>x == y
True
```

Якщо змінні x та y дорівнюють одна одній, то повертається значення True.

```
>>> not (x == y), not x == y
(False, False)
```

Через те, що перед виразом стоїть оператор not, вираз поверне False.

Круглі дужки можна не вказувати, оскільки оператор not має більш низький пріоритет виконання, ніж оператори порівняння.

3.3.5. Логічні оператори

and - логічне AND.

Приклад 15. Таблиця істинності оператора and

True and True	True
True and False	False
False and True	False
False and False	False

and над логічними змінними

```
>>>a = True; b = False
>>>a and b
False
```

and над логічними виразами

```
>>> 1 < 5 and 2 < 5 #True and True == True
True
>>> 1 > 5 and 2 < 5 #False and True == False
False
```

and над числами

```
# Якщо два числа, то повертається останнє число у виразі
>>> 10 and 20 # 10>0 та 20>0 результат 20
>>> 20 and 10 # 20>0 та 10>0 результат 10
>>>20  and -10# 20>0 та 10<0 результат -10
>>>-10 and 20 # -10<0 та 20>0 результат 20
# Якщо у виразі 0, то завжди повертається 0
>>>0   and 20# 0 та 20>0 результат 0
>>>20  and 0# 20>0 та 0 результат 0
```

or - логічне OR.

Приклад 3.16. Таблиця істинності оператора **or**

True or True	True
True or False	True
False or True	True
False or False	False

or над логічними змінними

```
>>>a = True; b = False
>>>a or b
True
```

or над логічними виразами

```
>>> 1 < 5 or 2 < 5 # True or True == True
True
>>> 1 < 5 or 2 > 5 # True or False == True
True
```

```
>>> 1 > 5 or 2 < 5 # False or True == True
True
>>> 1 > 5 or 2 > 5 # False or False == False
False
```

or над числами

```
# Два числа повертають перше число
>>> 10 or 20  # Результат 10
>>> -10 or 20 # Результат -10
# Одне число 0 - повертається число
>>> 0 or 20   # Результат 20
>>> -20 or 0  # Результат -20
# Два логічні значення символів
>>> [] or "s" # Результат 's'
>>> "" or "m" # Результат 'm'
```

Приклад 3.17.

```
>>> x1=input("Ввід x1=")
>>> x2=input("Ввід x2=")
>>> x3=input("Ввід x3=")
>>> x4=input("Ввід x4=")
>>> x1 == x2 and x2 != x3
>>> x1 == x2 or x3 == x4
```

Вираз поверне True тільки у випадку, якщо обидва вирази повернуть True:

Вираз поверне True, якщо хоча б один з виразів поверне True:

Приклад 3.18. Одночасно кілька умов:

```
>>> x = 10
>>> 1 < x < 20, 11 < x < 20
(True, False)
```

Оператори тотожності

Ці оператори перевіряють, чи є операнди однаковими (займають вони одну і ту ж позицію в пам'яті).

Оператор **is**

Якщо операнди тотожні, то повернеться **True.**

В іншому випадку **- False.**

Приклад 3.19.

```
>>> 2 is 20
False
>>> '2' is "2"
True
```

Оператор **is not**

Якщо операнди тотожні, то повернеться **False.**

В іншому випадку **- True.**

Приклад 3.20.

```
>>> 2 is not 20
True
>>> '2' is "2"
False
```

3.3.6. Пріоритет виконання операторів

У якій послідовності буде обчислюватися вираз?

$$x = 5 + 10 * 3/2$$

Це залежить від пріоритету виконання операторів. У цьому випадку послідовність обчислення виразу буде такою:

1. Число 10 буде помножено на 3, тому що пріоритет оператора множення вищий за пріоритет оператора додавання.

2. Отримане значення буде поділено на 2, тому що пріоритет оператора

ділення рівний пріоритету оператора множення (а оператори з рівними пріоритетами виконуються зліва направо), але вищий, ніж в оператора додавання.

3. До отриманого значення буде додане число 5, тому що оператор присвоювання = має найменший пріоритет.

4. Значення буде присвоєно змінній x.

```
>>> x = 5 + 10 * 3/2
>>> x
20.0
```

За допомогою дужок можна змінити послідовність обчислення виразу:

x = (5 + 10) * 3/2

Тепер порядок обчислень стане іншим:

1. До числа 5 буде додано 10.
2. Отримане значення буде помножено на 3.
3. Отримане значення буде поділено на 2.
4. Значення буде присвоєно змінній x.

```
>>> x = (5 + 10) * 3/2
>>> x
22.5
```

Оператори в порядку зменшення пріоритету

`()` – дужки

`**` – піднесення до степеню

`-x, +x, ~x` – унарный мінус, унарный плюс, двійкова інверсія,

`*, %, /, //` – множення (повторення), залишок від ділення, ділення, ділення з округленням униз.

`+, -` – додавання (конкатенація), віднімання.

`<<, >>` – двійкові зсуви.

& -двійкове AND.

^ -двійковий XOR.

| - двійкове OR.

=, +=, -=, *=, /=, //=, % =, **= - присвоювання.

Оператори порівняння в порядку зменшення пріоритету

<, >, <=, >=, =, ! =, <>, is, is not, in, not in.

Логічні оператори

not - логічне NOT.

and - логічне AND.

or - логічне OR.

3.4. Оператори розгалуження й цикли в мові Python

Програма мовою Python - це послідовність інструкцій, які записують з нового рядка або після крапки з комою.

Інструкція – це одиниця роботи (команда для комп'ютера), яка написана мовою Python.

Кінець рядка визначає кінець інструкції. Ніяких додаткових позначень кінця інструкції не потрібно.

Блок інструкцій – це група інструкцій, які виконуються підряд і починаються на однаковій відстані від початку рядка.

Вкладений блок інструкцій, - це блок, який знаходиться в середині іншого блока та виділяється величиною відступів від початку рядка.

Відступ від початку рядка може боти довільним. Рекомендований відступ складає 4 пробіли.

Структурні оператори – це оператори, які включають інші оператори та керують послідовністю їх виконання.

Існує дві групи структурних операторів:

1. Оператор розгалуження:

   if-elif-else - оператор

2. Оператори циклу:

   for – оператор

   while – оператор

3.4.1. Оператор розгалуження **if-elif-else**

Оператор розгалуження **if-elif-else** включає логічний вираз. В залежності від значення логічного виразу забезпечує виконання або невиконання окремого блоку інструкцій.

Існують різні рівні складності формату оператора.

Найпростіший формат оператора **if** :

```
if <Логічний вираз>:
```

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

Нагадаємо, що **Блок** – *це одна або більше інструкцій на мові Python, які записані підряд на однаковій відстані від початку рядка.*

```
if <Логічний вираз>:
```

$\xrightarrow{4пр}$ `<Інструкція 1>`

$\xrightarrow{4пр}$ `<Інструкція 2>`

$\xrightarrow{4пр}$ `<Інструкція 3>`

Приклад 3.21.

```
numvar = int(input("Write the number"))
firstvar, secondvar, thirdvar = 0,0,0
if numvar < 20:
```

```
    firstvar = numvar
    secondvar = numvar**2
    thirdvar = numvar**3
result = firstvar + secondvar + thirdvar
print("first_power=",numvar)
print("second_power=",secondvar)
print("third_power=",thirdvar)
print("Total=",result)
Write the number2
first_power= 2
second_power= 4
third_power= 8
Total= 14
```

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 3.22.

```
numvar = int(input("Write the number: "))
if numvar < 20:
    floor=numvar // 2
    remainder= numvar % 2
    second_power= numvar**2
else:
  print ("Your number is greater than 19")
  floor=-numvar // 2
  remainder= -numvar % 2
  second_power= -numvar**2

print("floor=",floor)
print("remainder= ",remainder)
print("second_power= ",second_power)
```

3.4.2. Синтаксис операторів

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

2. Ознакою кінця блоку є зсув першої за блоком 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.

4.Одна інструкція може розміщатися в одному рядку з ключовим словом

оператора.

5.Кілька інструкцій в одному рядку відділяються крапкою з комою.

Приклад 3.23.

```
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 3.24

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

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

Приклад 3.25

```
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 3.26.

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

3.4.3. 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 3.27

```
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 19")
```

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 3.28.

```
print("""Якою операційною системою ви користуєтесь?
1 - Windows 11
2 - Windows 10
3 - Windows 8
4 - Windows 7
5 - Інша""")
os = input("Введіть відповідне число: ")
if os == "1":
    print("Ви вибрали: Windows 11")
elif os == "2":
    print("Ви вибрали: Windows 10")
elif os == "3":
    print("Ви вибрали: Windows 8")
elif os == "4":
    print("Ви вибрали: Windows 7")
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.

3.4.4. 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 3.29.

```
print("""Якою операційною системою вы користуєтесь?
1 - Windows 11
2 - Windows 10
3 - Windows 8
4 - Windows 7
5 - Інша""")
os = input("Введіть відповідне число: ")
if os :
    if os == "1":
        print("Ви вибрали: Windows 11")
    elif os == "2":
        print("Ви вибрали: Windows 10")
    elif os == "3":
```

```
        print("Ви вибрали: Windows 8")
    elif os == "4":
        print ( "Ви вибрали: Windows 7")
    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 11
2 - Windows 10
З - Windows 8
4 - Windows 7
5 - Інша""")
os = input("Введіть відповідне число: ")
if not os:
    print("Ви не ввели число")
else:
    if os == "1":
        print("Ви вибрали: Windows 11")
    else:
        if os == "2":
            print("Ви вибрали: Windows 10")
        else:
            if os == "3":
                print("Ви вибрали: Windows 8")
            else:
```

```
                if os == "4":
                    print("Ви вибрали: Windows 7")
                else:
                    if os == "5":
                        print("Ви вибрали: інша")
                    else:
                        if os not in "12345":
                            print("Ми не визначили
вашу ОС")
```

Приклад 3.30.

```
>>> 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'
```

Ця конструкція приймає участь у генераторі списків.

3.4.5. Оператор циклу `for-in-else`

Існують різні рівні складності формату оператора for.

Оператор циклу for дозволяє виконувати ті самі інструкції багаторазово.

Оператор циклу for застосовують для:

1. перебору елементів послідовності.

   У цьому випадку змінній циклу почергово будуть присвоюватися значення елементів послідовності.

2. доступу до елементів послідовності по індексу

У цьому випадку використовуємо послідовність індексів, або ітератори, які видають таку послідовність.

3.4.6. Формат оператора циклу `for`

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

`for` <змінна циклу> **`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.

3.4.7. Конструкції оператора for

`for` <змінна циклу> **`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.;
- <Змінна циклу> – 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 3.31. Програма перебору букв у слові

```
for s in "Я вчуся програмувати":
    print(s, end=" ")
else:
    print ( "\nЦикл 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 3.32. 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

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 3.33. 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 3.34.Перебір словника другим способом

```
arr = {"x": 1, "y": 2, "z": 3}

for key in arr:
     print(key, arr[key])
```

Перебір словника із 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 3.35.

```
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 3.36.

```
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 3.37.

```
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 з зірочкою

Приклад 3.38.

```
arr = [[1,2],[3,4,5]]
#Список списків різної довжини
for x,y,*z in arr: # перебір з 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 з 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 3.39.

```
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]
```

Змінна i на кожній ітерації циклу містить лише копію значення поточного елемента списку.

Змінити у такий спосіб елементи списку не можна.

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 3.40.

```
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 3.41. Виведемо числа від 1 до 100:

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

Приклад 3.42. 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 3.43. Виведемо всі парні числа від 1 до 100:

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

3.4.8. Використання функції **range()**

В Python 2 функція **range()** повертає список чисел. В Python 3 функція **range()** повертає діапазон.

Щоб одержати список чисел, слід передати діапазон, що повернула функція **range()**, у функцію **list()**.

Приклад 3.44.

```
>>> obj = range(len([1, 2, 3]))
>>> obj
range(0, 3)
>>> obj[0],obj[1],obj[2] # Доступ по індексу
(0, 1, 2)
>>> obj[0:2]             # Отримання зрізу
range(0, 2)
>>> i=iter(obj)
>>> next(i),next(i),next(i)# Доступ з ітератором
(0, 1, 2)
>>> list(obj)   # Перетворення діапазону на список
[0, 1, 2]
```

```
>>> 1 in obj, 7 in obj # Перевірка на входження
значення
(True, False)
```

Діапазон підтримує два корисних методи:

index (<Значення>) – повертає індекс елемента, що має зазначене значення. Якщо значення не входить у діапазон, виконується виключення **ValueError.**

Приклад 3.45.

```
>>> obj = range(1, 5)
>>> obj.index(1), obj.index(4)
(0, 3)
>>> obj.index(5)
….Фрагмент опущений….
ValueError: 5 is not in range
```

Значення:	1	2	3	4
Індекс:	0	1	2	3

count (<Значення>) – повертає кількість елементів із зазначеним значенням. Якщо елемент не входить у діапазон, повертається значення 0.

Приклад 3.46.

```
>>> obj = range(1, 5)
>>> obj.count(1), obj.count(10)
(1,0)
```

3.4.9. Використання функції 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 3.47.

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 3.48

```
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')
```

3.4.10. Оператор циклу while

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

```
<Початкове значення лічильника>
while <Умова>:
    <Інструкції>
    <Збільшення лічильника>
[else:
    <Блок виконується за умови, що не використовується оператор break>]
```

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 3.49.

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 3.50.

Виведемо всі числа від 100 до 1.

```
i = 100
while i:
    print(i)
    i -= 1
```

1. Тут 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.

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

Приклад 3.51.

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

Результат: [2, 4, 6]

3.4.11. 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 3.52. Застосування оператора `continue`

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

Оператор **`break`**

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

Приклад 3.53 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 3.54. 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 )
```

Процес введення трьох чисел і одержання суми має такий вигляд (значення, введені користувачем, виділені напівжирним шрифтом):

Введіть слово 'stop' для одержання результату

Введіть число: **10**

0+10

Введіть число: **20**

0+10+20

Введіть число: **30**

0+10+20+30

Введіть число: **stop**

0+10+20+30 = 60

Приклад 3.55. Обчислювач

```
from math import *
print("""Виберіть операцію:
                1. Довжина кола
                2. Площа круга
                3. Об'єм кулі
                4. ВИХІД""")

while True:
    choice = input("ВВЕДІТЬ 1/2/3/4: ")
    if choice=='1' or choice=='2' or choice =='3':
        num = float(input("Введіть радіус: "))
        if choice == '1':
            print("Довжина кола =", 2*pi*num)

        elif choice == '2':
            print("Площа круга=", pi*num**2)

        elif choice == '3':
            print("Об'єм кулі =", 4/3*pi*num**3)

    elif choice=='4':
        print("Обчислювач закінчив роботу")
        break
    else:
        print("Ви ввели не те, що просили")
```

Приклад 3.56. Простий калькулятор

```
print("""Виберіть операцію:
                1.Додавання (+)
                2.Віднімання (-)
                3.Множення (x)
                4.Діленя (/)
                5.ВИХІД""")

while True:
    # Take input from the user
    choice = input("ВВЕДІТЬ 1 або 2 або 3 або 4 або
5: ")

    # Check if choice is one of the four options
    if choice in ('1', '2', '3', '4'):
        num1 = float(input("Введіть перше число: "))
        num2 = float(input("Введіть друге число: "))
        if choice == '1':
            print(num1, "+", num2, "=", num1+num2)

        elif choice == '2':
            print(num1, "-", num2, "=", num1-num2)

        elif choice == '3':
            print(num1, "*", num2, "=", num1*num2)

        elif choice == '4':
            if num2 != 0:
```

```
                print(num1, "/", num2, "=",
num1/num2)
            else:
                print("Ділити на нуль заборонено")
                continue
        elif choice == '5':
            break
    elif choice=='5':
        print("Калькулятор закінчив роботу")
        break
    else:
        print("Помилковий ввід даних")
```

3.5 Додаткові спопсоби вибору варіантів

3.5.1. Класичний спосіб вибору

```
  years=int(input("Введіть повну кількість прожитих
років: "))
if 0<=years< 8:
    print("Ви знаходитесь у першому періоді
дитинства")
elif 8<=years<12:
    print("Ви знаходитесь у другому періоді
дитинства")
elif 12<=years<16:
    print("У вас підлітковий вік")
elif 16<=years<21:
    print("У вас юнацький вік")
elif 21<=years<46:
```

```
    print("У вас молодий вік")
elif 46<=years<61:
    print("У вас середній вік")
elif 61<=years<76:
    print("У вас похилий вік")
elif 76<=years<91:
    print("Ви людина старшрого віку")
elif 91<=years:
    print("Ви довгожитель")
```

3.5.2. Вибір умови по ключу словника

```
temp =int(input("Введіть температуру: "))
d={temp < -20:   'Холодно',
-20 <= temp < 0: 'Прохолодно',
  0 <= temp < 15:'Зимно',
 15 <= temp < 25:'Тепло',
 25 <= temp:     'Жарко'}
print(temp,"градусів-",d[True], sep="")
```

Недолік підходу полягає у тому, що умови у ключах словника не повинні перекриватися. При перекриванні умов відповідь не буде однозначною.

3.5.3. Вибір варіанту по ключу словника (аналог **switch-case**)

```
month=float(input("Введіть номер місяця"))
switcher = {
        1: "Січень",
        2: "Лютий",
        3: "Березень",
        4: "Квітень",
```

```
        5: "Травень",
        6: "Червень",
        7: "Липень",
        8: "Серпень",
        9: "Вересень",
        10: "Жовтень",
        11: "Листопад",
        12: "Грудень"
    }
print(switcher.get(month, "Помилковий ввід"))
month=int(input("Введіть номер місяця"))
year=("Січень","Лютий","Березень","Квітень",
      "Травень","Червень","Липень","Серпень",
      "Вересень","Жовтень","Листопад","Грудень")
if 1<=month<=12:
    print(year[month-1])
else: print("Помилковий ввід даних")
```

Контрольні заритання до розділу 3.

1. Які типи операторів використовуються у мові Python?
2. Опишіть правила застосування арифметичних операцій.
3. Наведіть приклади застосування побітових операцій.
4. Основи застосування спеціалізованих операторів та операторів присвоювання.
5. Варіанти застосування оператра розгалуження
6. Застосування та формат операторів циклу.
7. Опишіть правила застосування операторів continue та break.

4. МЕТОДИ І ФУНКЦІЇ СТАНДАРТНИХ ТИПІВ ДАНИХ

4.1. Створення об’єктів числового типу

`int` – цілі числа. Розмір числа обмежений лише обсягом оперативної пам'яті:

Приклад: `1, 210, 34563`

`float` – дійсні числа;

Формат десяткового дробу: `1.1, 2.234 10.12, 34563.1`

Науковий формат:m**e**p-m-мантиса, p-порядок

Цілочисельна мантиса: `12e2=1200, 1e-1=0.1`

Дробова мантиса: `1.2e3=1200, 0.01e1=0.1`

Нормалізована мантиса: `0.12e4=1200, 0.1e2=10`

`complex` – комплексні числа

```
>>> 1+2j+2+3j
(3+5j)
```

Операції над числами різних типів повертають число, що має більш складний тип з типів, що брав участь в операції.

Співвідношення типів чисел при арифметичних операціях

1. Цілі числа мають найпростіший тип,
2. Далі йдуть дійсні числа.
3. Найскладніший тип – комплексні числа.

Правило перетворення

Якщо в операції беруть участь ціле число й дійсне, то ціле число буде автоматично перетворене в дійсне число, а потім виконана операція над дійсними числами.

Результатом цієї операції буде дійсне число.

`>>> 1+2.0` `3.0`	`>>> 2+0.2e1` `4.0`	`>>> 3.1+3+4j` `(6.1+4j)`

4.1.1. Створення об'єктів цілочисельного типу в десятковій системі числення

Приклад 4.1.

```
>>> x, y, z = 1,10,-80
>>> x,y,z
(1, 10, -80)
>>> p = 1,10,-80
>>> p
    (1, 10, -80)
>>> list(p)
[1, 10, -80]
>>> a="12345678"
>>> list(a)
['1', '2', '3', '4', '5', '6', '7','8']
```

Створення об’єктів за допомогою функції **`range()`**

Приклад 4.2.

```
>>> list(range(9))
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> tuple(range(1,9))
(1, 2, 3, 4, 5, 6, 7, 8)
>>> tuple(range(1,20,2))
(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
>>> tuple(range(50,1,-4))
(50,46,42,38,34,30,26,22,18,14,10,6,2)
```

4.1.2. Створення об'єкта цілочисельного типу у двійковій системі числення

Двійкові числа починаються з комбінації символів `0b` (або `0B`) і містять цифри 0 або 1.

Приклад 4.3.

```
>>> 0b111, 0b101
(7, 5)
>>> range(0b1010)
range(0, 10)
>>> list(range(0b10,0b1010))
[2, 3, 4, 5, 6, 7, 8, 9]
```

4.1.3. Створення об'єкта цілочисельного типу у вісімковій системі числення

Вісімкове число починається з нуля й наступної за ним латинської букви o (регістр не має значення) і містять цифри від 0 до 7:

Приклад 4.4.

```
>>>  0o0, 0o1, 0o7, 0o10, 0o11
(0, 1, 7, 8, 9)
>>> range(0o12)
range(0, 10)
>>> tuple(range(0o10,0o12))
(8, 9)
```

4.1.4. Створення об'єкта цілочисельного типу в шістнадцятковій системі числення

Шістнадцяткові числа починаються з комбінації символів 0x (або 0X) і можуть містити цифри від 0 до 9 і букви від A до F (регістр букв не має значення):

Приклад 4.5.

```
>>> 0x1,0x2, 0X9, 0xa, 0xf
(1, 2, 9, 10, 15)
>>> range(0xb)
range(0, 11)
>>> list(range(0xa,0xf))
[10, 11, 12, 13, 14]
```

4.1.5. Створення об'єкта дійсного типу

Дійсне число може містити крапку й (або) бути задане в науковому форматі (експонентній формі) з буквою E (регістр не має значення):

Приклад 4.6.

```
>>> x, y, z = 1.1,10.12,-80.455
>>> x,y,z
(1.1, 10.12, -80.455)
```

Зріз у послідовності

```
        0  1  2   3     4     5       6     7
>>> p = [1.1, 2,345, 0.23, 45.67, 11.678, 4.67, 11.56]
>>> p[1:3]- зріз від елемента 1 до елемента (3)
[2, 345]
>>> p = [1.1, 2,345, 0.23, 45.67, 11.678, 4.67, 11.56]
>>> p[2:7] – зріз від елемента 2 до елемента (7)
[345, 0.23, 45.67, 11.678, 4.67]
```

При виконанні операцій над дійсними числами слід враховувати обмеження точності обчислень.

Результат наступної операції може здатися дивним.

Приклад 4.7.

```
>>> 0.3 - 0.1 - 0.1 - 0.1
-2.7755575615628914e-17
>>> 0.6-0.2-0.4
-5.551115123125783e-17
```

Очікуваним був би результат 0.0, але, як видно з прикладу, ми одержали зовсім інше значення.

4.1.6. Використання функції **`Decimal i Fraction`**

Якщо необхідно виконувати операції з фіксованою точністю, то слід використовувати модуль `decimal`.

Перед виконанням таких операцій викликаємо:

```
from decimal import Decimal
```

Приклад 4.8.

```
from decimal import Decimal
a=Decimal("0.2")
b=Decimal("0.1")
n=a-b-b
print(n)
```

Результат:

```
0.0
```

Підтримка дробів виконується за допомогою модуля `fractions`.

При створенні дробу можна:

1. Указати два числа: чисельник і знаменник.
2. Указати одне число.
3. Указати рядок, що містить число, яке буде перетворено в дріб.

Приклад 4.9. Операція додавання дробів 4/5 та 3/5:

```
from fractions import Fraction
Rez=Fraction(4, 5)+Fraction(3, 5)
print(Rez)
print(type(Rez))
```

Результат:

```
7/5
<class 'fractions.Fraction'>
```

Приклад 4.10. Задати дріб ½ трьома способами

1.Задавання за допомогою дробу

```
>>> Fraction(1, 2)
Fraction(1, 2)
```

2.Задавання за допомогою рядка

```
>>> Fraction("0.5")
Fraction(l, 2)
```

3.Задавання за допомогою числа

```
>>> Fraction(0.5)
Fraction(l, 2)
```

4.Задавання нескорочуваного дробу

```
from fractions import Fraction
Rez=Fraction("0.57")
print(Rez)
Результат:
57/100
```

4.2. Операції і функції для числових типів

4.2.1. Арифметичні операції над дробами

Над дробами можна виконувати арифметичні операції, як і над звичайними числами.

Приклад 4.11. Виконати операції:

```
from fractions import Fraction
p1=Fraction(1, 2)
p2=Fraction(1, 4)
p3=p1+p2
print(p3)
p3=p1-p2
print(p3)
p3=p1*p2
print(p3)
p3=p1/p2
print(p3)
```

```
3/4
1/4

1/8

2
```

4.2.2. Операції з комплексними числами

Комплексні числа записуються у форматі:

<Дійсна частина>**+**<Уявна частина>**J**

Для задавання можна використовувати функцію `complex`.

Приклад 4.12.

```
>>> complex(2,3)
(2+3j)
```

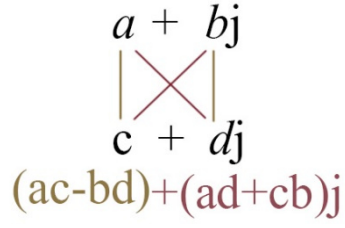
Клас комплексних чисел має кілька вбудованих методів.

Приклад 4.13.

```
>>> z = 2+3j
>>> z.real
2.0          # ціла частина
>>> z.imag
3.0          # уявна частина
>>> z.conjugate()
(2-3j)       # спряжене до 2+3j число
```

Приклад 4.14.

```
>>> complex(2,3)==2+3J
True
>>> 2+3J+4+7J
(6+10j)
>>> complex(2,3)+complex(4,7)
(6+10j)
>>> complex(2,3)-complex(4,7)
(-2-4j)
>>> complex(2,3)*complex(4,7)
(-13+26j)
>>> complex(2,3)/complex(4,7)
(0.4461538461538461-
0.03076923076923078j)
```



Правило множення: (a+bJ)*(c+dJ)

M=(ac-bd)+(ad+bc)J

M=(2*4-3*7)+(2*7+3*4)J

```
>>> complex(2,3)*complex(4,7)
```

```
(-13+26j)
```

Правило ділення:

Частка від ділення

числа $z_1 = a+bJ$ на число $z_2 = c+dJ$ визначається з виразу:

$$z=\frac{z_1}{z_2}=\frac{ac+bd}{c^2+d^2}+\frac{cb-ad}{c^2+d^2}J$$

4.2.3. Вбудовані функції для роботи із числами

Для роботи із числами будемо використовувати наступні вбудовані функції:

1. **int**([<Об'єкт>[,<Система числення>]])

Функція перетворює об'єкт у ціле число. У другому параметрі можна вказати систему числення перетвореного числа (значення за замовчуванням 10).

Приклад 4.15.

```
>>> int(7.5)
7
>>> int("71",10)
71
>>> int("0B1010",2)
10
>>> int("0o71",8)
57
>>> int("0xa",16)
10
```

2. **bin**(<Число>)

Функція перетворює двійкове, вісімкове, десяткове або шістнадцяткове число типу **int** у двійкове. Повертає рядкове представлення двійкового числа.

Приклад 4.16.

```
>>> bin(255), bin(1), bin(-45)
('0b11111111', '0b1', '-0b101101')
```

3. **oct**(<Число>)

Функція перетворює двійкове, вісімкове, десяткове або шістнадцяткове число типу **int** у вісімкове. Повертає рядкове представлення числа.

Приклад 4.17.

```
>>> oct(7), oct(8),  oct(64)
('0o7', '0o10', '0o100')
```

4. **hex**(<Число>)

Функція перетворює двійкове, вісімкове, десяткове або шістнадцяткове число типу int у шістнадцяткове. Повертає строкове представлення числа.

Приклад 4.18.

```
>>> hex(10), hex(16), hex(255)
('0xa', '0x10', '0xff')
```

5. **float**([<число або рядок>])

Перетворює ціле число або рядок у дійсне число

Приклад 4.19.

```
>>> float(7)
7.0
>>> float("7.1")
7.1
>>> float("inf")
inf
>>> float("nan")
nan
>>> float()
0.0
>>> float("inf")- float("inf")
nan
```

6.**round**(<Число>[,<Кількість знаків після крапки>])

Для чисел із дробовою частиною, меншою за 0.5, повертає число, округлене до найближчого меншого цілого, а для чисел із дробовою частиною, більшою за 0.5, повертає число, округлене до найближчого більшого цілого.

Якщо дробова частина дорівнює 0.5, то округлення проводиться до найближчого парного числа.

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

Приклад 4.20. Приклади округлення **round()**

```
>>> round(0.49),  round(0.50),  round(0.51)
(0, 0, 1) #0-парне число
>>> round(1.49),  round(1.50),  round(1.51)
(1, 2, 2)
>>> round(2.49),  round(2.50),  round(2.51)
(2, 2, 3)
>>> round(7.49),  round(7.50),  round(7.51)
(7, 8, 8)
```

У другому параметрі можна вказати бажану кількість знаків після коми. Якщо вона не зазначена, то використовується значення 0 (тобто число буде округлено до цілого):

Приклад 4.21.

```
>>> round(1.524,2),round(1.525,2),round(1.5555,3)
(1.52, 1.52, 1.556)
```

7. **abs**(<Число>) Повертає абсолютне значення:

```
>>> abs(-10),  abs(10),  abs(-12.5)
(10, 10, 12.5)
```

8. **pow**(<Число>,<Степінь>[,<Дільник>])

Підносить <Число> **до** <Степеня>

Приклад 4.21.

```
>>> pow(10,2),  10**2,  pow(3,3),  3**3
(100, 100, 27, 27)
```

Якщо зазначений третій параметр, то повертається залишок від ділення отриманого результату на значення цього параметра.

Приклад 4.22.

```
>>> pow(10,2,2),    (10**2)%2,    pow(3,3,2),    (3**3)%2
                (0, 0, 1, 1)
```

9. **max**(<Список чисел через кому>)

Повертає максимальне значення зі списку або послідовності. Повертається тип першого елемента у випадку однакових максимальних елементів.

Приклад 4.23.

```
>>> max (1,2,3)
3
>>>max (3,2,3,1)
3
>>> max (1,1.0)
1
>>> type(max (1.0,1))
int
>>> max (1.0,1)
1.0
>>> type(max (1.0,1))
float
```

max() function with iterables

>>> p = "strict"
>>> max(p)
't'
	>>>p=(1.0,0.5,0.8)
	>>> max(p)
	1 .0

>>>p=[34,11,34,124]
>>> max (p)
124
	>>> p = {False, True}
	>>> max (p)
	True

p=range(3) max(p) 2	p=["Python","C","Java"] max(p) 'Python'

Якщо елементами послідовності є рядки, тоді функція **max**() повертає рядок, що починається з символу з максимальним кодом

10. **min** (<список чисел через кому>)

min(<Послідовність>)

Мінімальне значення зі списку. Повертається тип першого елемента у випадку однакових мінімальних елементів

Приклад 4.24.

```
>>>min(1,2,3)
1
>>>min(3,2,3,1)
1
>>>min(1,1.0)
1
>>>min(1.0,1)
1.0
>>>min("c","a","m")
'a'
```

min() function with iterables

>>> p="string" >>>min(p) 'g' >>>p = "Tt" >>>min(p) 'T'	>>>p=(4,5,6,7,8,3) >>>min(p) 3
>>>p=[34,11.34,124] >>> m in (p) 11.34	>>>p={False, True } >>> m in(p) False
p=range(3) min(p) 0	p=["Python","JavaScript"] min(p) 'JavaScript'

Якщо елементами послідовності є рядки, тоді функція **min()** повертає рядок, що починається з символу з мінімальним кодом

Приклад 4.25.

```
#У списку знаходимо максимальне значення ключа
>>>max({"a":2016, "b":"London", "c": "West"})
'c'
#У списку знаходимо значення, на яке вказує максимальний ключ
>>>  p = {"a":2016, "b":"London", "c": "West"}
>>> dm =max(p)
>>> print(p[dm])
West
#У списку знаходимо значення, на яке вказує мінімальний ключ
>>> dm = min(p)
>>> print(p[dm])
2016
```

11. **sum** (<Послідовність>[, <Початкове значення>])

Правила роботи функції **sum()**

Функція повертає суму значень елементів послідовності (наприклад: списку, кортежу) плюс <**Початкове значення**>.

Якщо другий параметр не зазначений, **початкове значення** вважають рівним 0.

Якщо послідовність пуста, то повертається значення другого параметра.

Приклад 4.26.

```
>>> sum((10,20,30,40))  #Сума елементів
100
>>> sum ([10,20,30,40]) #Сума елементів
100
>>> sum([10,20,30,40],2)#Початкове значення
102
>>> sum([], 2)  # Пуста послідовність
2
>>> sum([])# Пуста послідовність
```

```
0
sum("a","b") #Помилка
TypeError: sum() can't sum strings [use ''.join(seq)
instead]
```

Функція sum *з послідовностями*

Кортеж

```
>>> p=(1,2,3,4,5,6,7,8,9)
>>> sum(p)
45
>>> sum(p,3)
48
```

Список:	**Множина:**
>>> p = [11,12,15,17]	**>>>** p = {67,11,78,12}
>>> sum(p)	**>>>** sum(p)
55	168
>>> sum(p,5)	**>>>** sum(p,2)
60	170

12. **divmod**(x, y)

повертає кортеж з двох значень (x **//**y, x **%** y):

Приклад 4.27.

```
>>> divmod(13, 2) #13 == 6 *2+ 1
(6, 1)
>>> 13//2, 13%2
(6,1)
>>> divmod(13.5,2.0) #13.5==6.0*2.0+1.5
   (6.0, 1.5)
>>> 13.5//2.0, 13.5 % 2.0
   (6.0, 1.5)
```

Вбудовані функції для комплексних чисел:

$\sqrt{3^2+4^2}=5$

```
>>> abs(3 + 4j)
    5.0
```

(3+4J)*(3+4J)

```
>>> pow(3 + 4j, 2)
(-7+24j)
```

4.2.4. Модуль **math**. *Математичні функції*

Модуль **math** надає додаткові функції для роботи із числами, а також стандартні константи. Перш ніж використовувати модуль, необхідно підключити його за допомогою інструкції:

```
import math
```

або

```
from math import *
```

Примітка

Для роботи з комплексними числами необхідно використовувати модуль cmath

```
import cmath # для комплексних чисел
import math # для дійсних чисел
```

Стандартні константи модуля **math**

```
import math
from decimal import Decimal
print(math.pi)# pi — повертає число π :
print(Decimal.from_float(math.pi))
```

Результат:

3.141592653589793

3.14159265358979311599796346854418516159057617187 5

print(math.e)# e — повертає значення константи e:

```
print(Decimal.from_float(math.e))
```

Результат:

2.718281828459045

2.718281828459045090795598298427648842334747314453125

Основні функції для роботи з числами з модуля **math.**

Аргумент задаємо в радіанах

- **sin**(), **cos**(), **tan**( ) — стандартні тригонометричні функції (синус, косинус, тангенс). Значення вказується в радіанах;
- Функція повертає тип даних float
- **asin**(), **acos**(), **atan**() — обернені тригонометричні функції (арксинус, арккосинус, арктангенс). Значення повертається в радіанах;
- Функція повертає тип даних float
- **degrees**() — перетворює радіани в градуси:
- Функція повертає тип даних float

Приклад 4.28.

```
>>> math.degrees(math.pi)
180.0
```

- **radians**() — перетворює градуси в радіани:

Приклад 4.29.

```
>>> math.radians(180.0)
3.141592653589793
```

exp() — експонента.

Приклад 4.30.

```
>>> import math
>>> math.exp(1)
 2.718281828459045
```

log(<число>[,<База>]) — логарифм по заданій базі. Якщо база не зазначена, обчислюється натуральний логарифм (по базі e)

log10() – десятковий логарифм;

log2() – логарифм по базі 2;

Функції повертають тип `float`

`sqrt``()` – квадратний корінь:

```
>>> math.sqrt(100), math.sqrt(25)
(10.0, 5.0)
```

- **`ceil`**`()` — значення, округлене до найближчого більшого цілого:

```
>>>import math
>>>math.ceil(6.50)
7
>>>import math
>>>math.ceil(6.51)
7
>>>import math
>>>math.ceil(-6.51)
-6
```

- **`floor`**`()` — значення, округлене до найближчого меншого цілого:

```
>>> math.floor(5.49)
5
>>> math.floor(5.50)
5
>>> math.floor(5.51)
5
```

- **`pow`**`(<Число>,<Степінь>)` —підносить<Число> до <Степеня>:
- Функція повертає тип даних `float`

```
>>> math.pow(10,2), 10**2,
(100.0, 100)
>>> math.pow(3,3), 3**3
(27.0, 27)
```

- **`fabs`** `( )` — абсолютне значення:

Функція повертає тип даних `float`

```
>>> math.fabs(10)
```

```
10.0
>>>math.fabs(-10
10.0
>>> math.fabs(-12.5)
12.5
```

♦ **fmod**() — залишок від ділення:

```
>>> math.fmod(10,5), 10%5,
(0.0,0)
>>>math.fmod(10,3), 10%3
(1.0,1)
```

♦ **factorial** () — факторіал числа:

```
>>>math.factorial(5)
120
>>>math.factorial(6)
720
```

♦ **fsum**(<Список чисел>) — повертає точну суму чисел із заданого списку:

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

Примітка

У цьому розділі ми розглянули тільки основні функції. Щоб одержати повний список функцій, звертайтеся до документації по модулю **math.**

4.2.5. Модуль **random.** *Генерація випадкових чисел*

Модуль **random** дозволяє генерувати випадкові числа. Перш ніж використовувати модуль, необхідно підключити його за допомогою інструкції:

```
import random
```

або

```
from random import *
```

Основні функції модуля **random**:

- **random**() — псевдовипадкове число від 0.0 до 1.0:

```
>>> import random
>>>random.random()
0.9753144027290991
>>>random.random()
0.5468390487484339
>>>random.random()
  0.1301505805476736
```

- **seed**([<параметр>][,version=2])

налаштовує генератор випадкових чисел на нову послідовність. Якщо перший параметр не зазначений, за базу для випадкових чисел буде використаний системний час. Якщо значення першого параметра буде однаковим, то генерується однакове число:

```
  >>> random.seed(10)
  >>> random.random()
  0.5714025946899135
  >>>  random.seed(10)
  >>> random.random()
  0.5714025946899135
```

- **uniform**(<початок>,<кінець>) — повертає псевдовипадкове дійсне число в діапазоні від <Початок> до <Кінець>:

```
  >>> random.uniform(0,10)
  9.965569925394552
  >>> random.uniform(0,10)
  0.4455638245043303
```

- **randint** (<початок>, <кінець>) — повертає псевдовипадкове ціле число в діапазоні від <Початок> **до** <Кінець>:

```
>>> random.randint(0,10)
4
>>> random.randint(0,10)
10
```

- **randrange**([<початок>,]<кінець>[,<Крок>]) повертає випадковий елемент із числової послідовності. Параметри аналогічні параметрам функції range(). Можна сказати, що функція randrange «за кадром» створює діапазон, з якого й будуть вибиратися випадкові числа, що повертаються:

```
>>> random.randrange(10)
5
>>> random.randrange(0,10)
2
>>> random.randrange(0,10,2)
6
```

- **choice**(<послідовність>) — повертає випадковий елемент із заданої послідовності (рядка, списку, кортежу):

```
>>> import random
>>>random.choice("string")
't'
>>>random.choice(["s","t","r"])#список
'r'
>>>random.choice(("s","t","r"))#кортеж
't'
>>>random.choice(["one","two","three","four"])
'two'
```

- **shuffle**(<список) — перемішує елементи списку випадковим чином. Дана функція жодного результату не повертає.

Приклад 4.30.

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> random.shuffle(arr)
>>> arr
[8, 6, 9, 5, 3, 7, 2, 4, 10, 1]
```

♦ **sample**(<Послідовність>,<Кільк.елементів>) — повертає список із зазначеної кількості елементів, які будуть обрані випадковим чином із заданої послідовності. За послідовність можна взяти будь-які об'єкти, що підтримують ітерації.

Приклад 4.31

```
>>> import random
>>> random.sample("string", 2)
  ['i','r']
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
[7,10]
>>> arr #сам список не змінюється
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample((1, 2, 3, 4, 5, 6, 7), 3)
[6, 3, 5]
>>> random.sample(range(300), 5)
[126, 194, 272, 46, 71]
```

4.2.6. Приклад створення генератора паролів довільної довжини

Для цього додаємо в список arr усі дозволені символи, а далі в циклі одержуємо випадковий елемент за допомогою функції **choice()**. За замовчуванням буде видаватися пароль з 8 символів.

Приклад 4.32.

```
import random # Підключаємо модуль randorn
def passw_generator(count_char=8):
 arr = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j
', 'k', '1', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
```

```
'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E',
'F', 'G', 'H', 'I', 'J', 'K', 'L','M', 'N', 'O', 'P',
'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'У','Z', '1',
'2', '3', '4', '5', '6', '7', '8', '9', '0']
 passw = ""
 for i in range(count_char):
     passw+=random.choice(arr)
 return passw
# Викликаємо функцію
ps=passw_generator(10)
print("PASSWORD:",ps)
```

4.2.7. Гра орел-решка

Приклад 4.33.

```
import random
print ("""
Вітаємо тебе у грі. Це гра в рел-решка, і ти будеш тим,
хто зможе вибрати відповідь. Давайте почнемо!
""")
cont = "1" # Початкове значення кількості ігор
while(cont == "1"):
    print("~-~-" * 10)
    theirGuess = input("Яка твоя здогадка? :   ")
    ourAnswer = random.choice(["Орел","Решка"])
    if ourAnswer == "Орел":
        notCorrectAnswer = "Решка"
    else:
        notCorrectAnswer = "Орел"
    if theirGuess == ourAnswer:
        print("Ти вгадав!")
    elif theirGuess != notCorrectAnswer and ourAnswer:
```

```
        print("Ти не вгадав! Спробуй знову орел та
решка!")
    else:
        print("Ти не вгадав! Спробуй вгадати наступний
раз!")

    cont = input("Хочеш продовжити? Введи 1 якщи Так,
введи 0 якщо Ні: ") #Зробіть вибір користувача щодо
того, запускати гру знову чи ні
    while (cont != "0" and cont != "1"):
        cont = input("Спробуй ще раз. Введи 1 якщо так,
введи 0 якщо Ні: ")
```

4.2.8. Гра в кості

Приклад 4.34.

```
import random
PlayerOne = "Марія"
PlayerTwo = "Іван"
MaryScore = 0
IvanScore = 0
# У кожного кубика шість значень
diceOne = [1, 2, 3, 4, 5, 6]
diceTwo = [1, 2, 3, 4, 5, 6]
def playDiceGame():
    """Обидва гравці, Марія і Іван, кидають кубик з
використанням методу  shuffle"""
    for i in range(5):
        # Трясемо кості 5 разів
        random.shuffle(diceOne)
        random.shuffle(diceTwo)
    firstNumber = random.choice(diceOne)  #
```

```
*Використання методу choice для вибору випадкового значення*
    SecondNumber = random.choice(diceTwo)
    return firstNumber + SecondNumber
print("Гра в кості використовує модуль random\n")
# Граємо три рази
for i in range(3):
    # визначемо, хто буде грати в кості першим
    IvanTossNumber = random.randint(1, 100)  # генерація випадкового числа від 1 до 100, включаючи 100
    MaryTossNumber = random.randrange(1, 101, 1) #генерація випадкового числа от 1 до 100, не включаючи 101
    if (IvanTossNumber > MaryTossNumber):
        print("Іван виграв жеребкування.")
        IvanScore = playDiceGame()
        MaryScore = playDiceGame()
    else:
        print("Марія виграла жеребкування.")
        MaryScore = playDiceGame()
        IvanScore = playDiceGame()

    if (IvanScore > MaryScore):
        print("Іван виграв гру в кості. Фінальний рахунок Івана:", IvanScore, "Фінальний рахунок Марії:", MaryScore, "\n")
    else:
        print("Марія виграла гру в кості. Фінальний рахунок Марії:", MaryScore, "Фінальний рахунок Івана:", IvanScore, "\n")
```

4.3. Створення рядків і робота з рядками

Приклад 4.35.

```
s=input('Write the number')
```

```
a=0
for i in s:
    for j in (range(10)):
        if i==str(j):a+=1
if a == len(s): print("Number")
else: print("Not number")
```

Загальна характеристика рядків в Python

Рядки – упорядковані послідовності символів.

```
a="Мій рядок"
for i in a:
    print (i,"-",a.index(i), end=(", "))
```

Результат: М - 0, і - 1, й - 2, - 3, р - 4, я - 5, д - 6, о - 7, к - 8,

1. Довжина рядка обмежена лише обсягом оперативної пам'яті комп'ютера.

2. Рядки підтримують:

1) доступ до елемента по індексу,

2) одержання зрізу,

3) конкатенацію (оператор +),

4) повторення (оператор *),

5) перевірку на входження (оператори in та not in).

Рядки є незмінюваними типами даних.

Тому практично всі строкові методи як значення повертають новий рядок.

```
s="Line"
m=s
print(m is s)
s=s+"1"
print(m is s)
```

Результат

True

False

При використанні невеликих рядків це не приводить до проблем

При роботі з більшими рядками можна зіткнутися з проблемою нестачі пам'яті.

Приклад 4.36. Спроба змінити символ по індексу

Можна одержати символ по індексу, але змінити не можна.

```
>>> s="Python"
>>> s[0]
'P'
>>> s[0]="J"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

У деяких мовах програмування кінцем рядка є нульовий символ.

У мові Python нульовий символ може бути розташований всередині рядка.

4.3.1. Основне поняття про кодову сторінку (code page)

Кодова сторінка – це таблиця, яка ставить у відповідність кожному символу його числове значення.

Характеристики кодових сторінок:

1. Підтримують певні мови на визначених ОС.
2. Один код кодової сторінки представляє тільки один символ.
3. Кожна кодова сторінка має свою унікальну назву.
4. Кожна ОС працює з визначеним набором кодових сторінок.
5. Існує кілька сотень кодових сторінок, які використовують ОС та системи передачі даних.

Набори символів

Single-Byte Character Set (SBCS) – 8-розрядні кодування (латиниця, кирилиця, грецькі, арабські, єврейські та тайські символи)

Double-Byte Character Set (DBCS) – 16 розрядні кодування включають також китайські та японські символи.

American Standard Code for Information Interchange

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

. Кодовая страница CP1251

128	Ъ	10000000	160	"	10100000	192	А	11000000	224	а	11100000
129	Ѓ	10000001	161	Ў	10100001	193	Б	11000001	225	б	11100001
130	‚	10000010	162	ў	10100010	194	В	11000010	226	в	11100010
131	ѓ	10000011	163	J	10100011	195	Г	11000011	227	г	11100011
132	„	10000100	164	¤	10100100	196	Д	11000100	228	д	11100100
133	…	10000101	165	Ґ	10100101	197	Е	11000101	229	е	11100101
134	†	10000110	166	¦	10100110	198	Ж	11000110	230	ж	11100110
135	‡	10000111	167	§	10100111	199	З	11000111	231	з	11100111
136	€	10001000	168	Ё	10101000	200	И	11001000	232	и	11101000
137	‰	10001001	169	©	10101001	201	Й	11001001	233	й	11101001
138	Љ	10001010	170	Є	10101010	202	К	11001010	234	к	11101010
139	‹	10001011	171	«	10101011	203	Л	11001011	235	л	11101011
140	Њ	10001100	172	¬	10101100	204	М	11001100	236	м	11101100
141	Ќ	10001101	173		10101101	205	Н	11001101	237	н	11101101
142	Ћ	10001110	174	®	10101110	206	О	11001110	238	о	11101110
143	Џ	10001111	175	Ы	10101111	207	П	11001111	239	п	11101111
144	ђ	10010000	176	°	10110000	208	Р	11010000	240	р	11110000
145	'	10010001	177	±	10110001	209	С	11010001	241	с	11110001
146	'	10010010	178	І	10110010	210	Т	11010010	242	т	11110010
147	""	10010011	179	ц	10110011	211	У	11010011	243	у	11110011
148	""	10010100	180	ґ	10110100	212	Ф	11010100	244	ф	11110100
149	•	10010101	181	µ	10110101	213	Х	11010101	245	х	11110101
150	–	10010110	182	¶	10110110	214	Ц	11010110	246	ц	11110110
151	—	10010111	183	·	10110111	215	Ч	11010111	247	ч	11110111
152	~	10011000	184	ё	10111000	216	Ш	11011000	248	ш	11111000
153	™	10011001	185	№	10111001	217	Щ	11011001	249	щ	11111001
154	љ	10011010	186	є	10111010	218	Ъ	11011010	250	ъ	11111010
155	›	10011011	187	»	10111011	219	Ы	11011011	251	ы	11111011
156	њ	10011100	188	j	10111100	220	Ь	11011100	252	ь	11111100
157	ќ	10011101	189	S	10111101	221	Э	11011101	253	э	11111101
158	ћ	10011110	190	s	10111110	222	Ю	11011110	254	ю	11111110
159	џ	10011111	191	В	10111111	223	Я	11011111	255	я	11111111

4.3.2. Строкові типи, підтримувані в Python 3

str - Unicode-рядок.

При виводі Unicode-рядок можна перетворити в послідовність байтів у будь-якому кодуванні:

Приклад 4.37.

```
>>> s="рядок"
>>> type(s)
<class 'str'>
>>>c=s.encode(encoding="cp1251")
>>>type(c)
<class 'bytes'>
>>>print(c)
b'\xf0\xff\xe4\xee\xea'
     р  я  д  о  к
>>> s.encode(encoding="utf-8")
b'\xd1\x80\xd1\x8f\xd0\xb4\xd0\xbe\xd0\xba'
```

Тип **bytes.** Незмінювана послідовність байтів.

1.Кожний елемент послідовності може зберігати ціле число від 0 до 255, яке може позначати код символу, або просто дані.

2.Об'єкт типу **bytes** підтримує ряд методів спільних з типом `str`

3. Латиниця виводиться символами, кирилиця виводиться кодами.

4. Доступ по індексу повертає ціле число, а не символ.

Приклад 4.38.

```
>>> m = bytes("стр str", "cp1251")
>>> m
b'\xf1\xf2\xf0 str'
>>> m[0], m[5], m[0:3], m[4:7]
(241, 116, b'\xf1\xf2\xf0', b'str')
>>> m[0], hex(m[0])
(241, '0xf1')
```

```
>>> m[5],hex(m[5])
(116, '0x74')
```

4.3.3. Зберігання символів у bytes

Об'єкт типу **bytes** може містити як однобайтні, так і багатобайтні символи.

Функція **len()** для типів **bytes** і **str.**

Функція **len ()** повертає кількість байтів у даних типу **bytes**

Функція **len ()** повертає кількість символів типу **str**

Приклад 4.39.

```
>>> len("рядок")
5
m = bytes("рядок", "cp1251")
>>> len (m)
5
m = bytes("рядок", "utf-8")
>>> len (m)
10
```

Тип **bytearray.** *Змінювана послідовність байтів.*

1.Тип **bytearray** аналогічний типу **bytes.**

2.Тип **bytearray** дозволяє змінювати елементи по індексу.

3.Містить методи, що дозволяють додавати й видаляти елементи.

Приклад 4.40.

```
s = bytearray("str", "cp1251")
s[0] = 49; # Можна змінити символ
print(s)
s.append(55) # Можна додати символ
print(s)
Результат
bytearray(b'1tr')
```

bytearray(b'**1tr7**')

1. У всіх випадках, коли йдеться про текстові дані, слід використовувати тип **str.** Саме цей тип ми будемо називати словом «**рядок**».

2.Типи **bytes** і **bytearray** слід задіяти для запису бінарних даних – наприклад, зображень, а також для проміжного зберігання текстових даних.

4.3.4. Створення і використання рядків

Способи створення рядка:

- за допомогою функції

str ([<Об'єкт>[,<Кодування>[,<Обробка помилок>]]])

1. Якщо зазначений тільки перший параметр, то функція повертає строкове представлення будь-якого об'єкта.

2. Якщо параметри не зазначені взагалі, то повертається порожній рядок.

```
>>>str()
''
```

Приклад 4.41.

```
>>> str([1, 2]), str((3, 4)), str({"x": 1})
('[1, 2]', '(3, 4)', "{'x': 1}")
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0")
"b'\\xf1\\xf2\\xf0\\xee\\xea\\xe0'"
```

Перетворення об'єктів типу **bytes**

Щоб одержати з об'єктів типу bytes і bytearray саме рядок, слід указати кодування в другому параметрі:

Приклад 4.42

```
>>> str(b"\xf0\xff\xe4\xee\xea", "cp1251")
'рядок'
```

У третьому параметрі функції **str** можуть бути зазначені значення

"strict" (при помилці виконується виключення Unicodedecodeerror – значення за замовчуванням),

"replace" (невідомий символ заміняється символом, що має код \ufffd)

"ignore" (невідомі символи ігноруються):

Приклад 4.43. Застосування третього параметра

```
>>> obj1 = bytes("рядок1", "utf-8")
>>> obj2 = bytearray("рядок2", "utf-8")
>>> str(obj1, "utf-8")
'рядок1'
>>>str(obj2, "utf-8")
'рядок2'
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str (obj1, "ascii", "ignore")
'1'
```

Спеціальні символи

Спеціальні символи – це комбінації знаків, що позначають службові символи або символи, що не друкуються.

\n - перевід рядка;

\r - повернення каретки;

\t - знак табуляції;

\v - вертикальна табуляція(виводить пробіл);

\a – дзвінок(виводить пробіл);

\b – вибій (видаляє попередній символ);

\f – перевід (зміна) формату;

\0 – нульовий символ (не є кінцем рядка);

\" - лапки;

\' - апостроф;

\N - вісімкове значення N. Наприклад, \74 відповідає символу **<**;

\xN - шістнадцяткове значення N. Наприклад,

\x6a відповідає символу j;

\\ - зворотний слеш;

\uxxxx- 16-бітний символ **Unicode.**

\uxxxxxxxx - 32-бітний символ **Unicode**, де x позначає шістнаднадцяткову цифру (4 двійкові розряди)

Наприклад, \u0004 відповідає російській букві к;

Апострофи й подвійні лапки:

Приклад 4.45.

```
>>> 'рядок', "рядок", '"x":5', "'x':5"
('рядок', 'рядок', '"x":5', "'x':5")
>>> print ('рядок1\nрядок2')
рядок1
рядок2
```

У деяких мовах програмування (наприклад, у PHP) рядок в апострофах відрізняється від рядка в лапках тим, що всередині апострофів спеціальні символи виводяться як є, а всередині лапок вони інтерпретуються. У мові Python жодної відмінності між рядком в апострофах і рядком у лапках немає.

Правила роботи з лапками

1. Якщо рядок містить лапки, то його краще «взяти» в апострофи, і навпаки.
2. Усі спеціальні символи в таких рядках інтерпретуються.
3. Послідовність символів \n перетвориться в символ нового рядка.
4. Щоб спеціальний символ виводився як є, його необхідно екранувати за допомогою слеша: \\n

Приклад 4.46. Варіанти екранування слешем.

```
>>> print("Рядок1\\nрядок2")
Рядок1\nрядок2
>>> print('Рядок1\nрядок2')
Рядок1
рядок2
```

Лапки всередині рядка в лапках і апостроф всередині рядка в апострофах також необхідно екранувати за допомогою захисного слеша:

Приклад 4.47.

```
 >>>"\"x\": 5", '\'x\': 5'
('"x": 5', "'x': 5")# Але апостроф в лапках
>>> "'x': 5", '"x": 5'#і лапки в апострофах
("'x': 5", '"x": 5')  # не екранують
```

Особливості переходу на новий рядок

1. «Взяти» об'єкт в одинарні лапки (або апострофи) на декількох рядках не можна. Перехід на новий рядок викличе синтаксичну помилку:

Приклад 4.48.

```
>>> "stri
ng"
Syntaxerror: EOL while scanning string literal
```

2. Щоб розташувати об'єкт на декількох рядках, слід:

А) перед символом переводу рядка вказати символ \, Б) помістити два рядки всередині дужок,

В) використовувати конкатенацію всередині дужок.

Приклад 4.49. Способи задавання багаторядкового тексту

```
>>> "string1\
string2" #після \ не повинно бути жодних символів
'string1string2'
>>> ("string1"
"string2")          # Неявна конкатенація рядків
'string1string2'
>>> ("string1" +
"string2")          # Явна конкатенація рядків
'string1string2'
```

Екранування слеша наприкінці рядка

Якщо наприкінці рядка розташований символ \, то його необхідно екранувати, інакше буде виведене повідомлення про помилку:

Приклад 4.50.

```
print("http:string\")
```

Результат:

```
Syntaxerror: EOL while scanning string literal
```

```
print("string\\")
```

Результат:

```
string\
```

Рядки з потроєними апострофами й лапками

1. Рядки між потроєними апострофами й потроєними лапками можна розмістити на декількох рядках.

'''a

b'''

2. У таких рядках допускається одночасно використовувати й лапки, і апострофи без необхідності їх екранувати.

```
""" "a"b'c"""
```

3. В решті випадків такі об'єкти еквівалентні рядкам в апострофах і лапках. Усі спеціальні символи в таких рядках інтерпретуються.

```
print(""" "a"\nb'c""")
```

Результат:

```
"a"
b'c'
```

Приклад 4.51.

```
print('''Рядок1
Рядок2''')
```

Результат виконання:

```
Рядок1
Рядок2
```

```
print ("""Іван
Марія""")
Результат виконання:
Іван
Марія
```

Рядок документування

Якщо рядок не присвоюється змінній, то він вважається рядком документування. Такий рядок зберігається в атрибуті `__doc__` того об'єкта, у якому розташований. Як приклад створимо функцію з рядком документування, а потім виведемо вміст рядка:

```
def test():
    """Це опис функції"""
    pass
print(test.__doc__)
```

Результат:

```
Це опис функції
```

Оскільки вирази всередині таких рядків не виконуються, то потроєні лапки (або потроєні апострофи) дуже часто використовуються для коментування більших фрагментів коду на етапі налаштування програми!!!

Рядок з модифікатором r

Якщо перед рядком розмістити модифікатор **r**, то спеціальні символи усередині рядка виводяться як є. Наприклад, символ **\n** не буде перетворений у символ переводу рядка. Іншими словами, він буде вважатися послідовністю двох символів: \ і **n**:

Приклад 4.52.

```
>>> print("Рядок1\nрядок2")
Рядок1
Рядок2
>>> print(r"Рядок1\nрядок2")
Рядок1\nрядок2
```

```
>>> print(r"""Рядок1\nрядок2""")
Рядок1\nрядок2
```

4.3.5. Застосування неформатованих рядків з модифікатором r

Такі неформатовані рядки зручно використовувати в шаблонах регулярних виразів, а також при вказівці шляху до файлу або каталогу:

Приклад 4.53.

```
>>>print(r"C:\Python39-4\lib\site-packages")
C:\Python39-4\lib\site-packages
```

Якщо модифікатор не вказати, то всі слеші при вказівці шляху необхідно екранувати:

Приклад 4.54.

```
>>>print("C:\\Python39-4\\lib\\site-packages")
C:\Python39-4\lib\site-packages
```

Слеш наприкінці рядка при використанні модифікатора r

Якщо наприкінці неформатованого рядка розташований слеш, то його необхідно екранувати.

Приклад 4.55.

```
>>> print(r"C:\Python39-4\lib\nigtwish\")
  File "<input>", line 1
    print(r"C:\Python39-4\lib\nightwish\")
                                         ^
Syntaxerror: EOL while scanning string literal
>>> print(r"C:\Python39-4\lib\nightwish\\"[:-1])
C:\Python39-4\lib\nightwish\
```

Інші способи забрати слеш наприкінці рядка

Щоб позбутися зайвого слеша, можна використовувати операцію конкатенації рядків, звичайні рядки або **вилучити слеш** явно.

Приклад 4.56.

```
# Конкатенація(виділяємо в окремий рядок)
>>> print("C:\Python39-4\lib\site-packages" + "\\")
```

```
C:\Python39-4\lib\site-packages\
# Звичайний рядок
>>> print("C:\\Python39-4\\lib\\site-packages\\")
C:\Python39-4\lib\site-packages\
# Видалення слеша
>>> print("C:\Python34\lib\site-packages\\"[:-1])
C:\Python34\lib\site-packages\
```

Використання слеша без спецсимволів

Якщо після слеша немає символа, який разом зі слешем інтерпретується як спецсимвол, то слеш зберігається в складі рядка:

Приклад 4.57.

```
>>> print("Цей символ \не спеціальний:")
Цей символ \не спеціальний
```

Проте, краще екранувати слеш явно:

Приклад 4.58.

```
>>> print("Цей символ \\не спеціальний:")
Цей символ \не спеціальний
```

4.3.6. Операції з рядками

1. Рядки є послідовностями. Тому вони підтримують

- доступ до елемента по індексу,
- одержання зрізу,
- конкатенацію,
- повторення,
- перевірку на входження.

Розглянемо ці операції докладно.

Доступ до елемента по індексу

До будь-якого символу рядка можна звернутися як до елемента списку – достатньо указати його індекс у квадратних дужках. Нумерація починається з нуля:

Приклад 4.59. Доступ по індексу

```
>>> s = "Python"
>>> s[0], s[1], s[2], s[3], s[4], s[5]
('P', 'y', 't', 'h', 'o', 'n')
```

Якщо символ, відповідний до зазначеного індексу, відсутній у рядку, то виконується виключення **Indexerror:**

Приклад 4.60. Індекс за межами рядка

```
>>> s = "Python"
>>> s[10]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
Indexerror: string index out of range
```

Доступ по від'ємному індексу

Як індекс можна вказати від'ємне значення. У цьому випадку зсув буде вестись від кінця рядка, а точніше – щоб одержати додатний індекс, віднімається від довжини рядка:

Приклад 4.61. (6-1=5)

```
>>> s = "Python"
>>> s[-1], s[len(s)-1]
('n', 'n')
```

Символ у рядку по індексу змінити не можна

Оскільки рядки відносяться до незмінних типів даних, змінити символ по індексу **не можна**:

Приклад 4.62.

```
>>> s = "Python"
>>> s [0] = "J" # Змінити рядок не можна
TypeError: 'str' object does not support item
assignment
>>> del s[0]
TypeError: 'str' object doesn't support item deletion
```

```
>>> s ="J"+s[1]+s[2]+s[3]+s[4]+s[5]
>>> s
'Jython'
```

Операція добування зрізу по рядку

Щоб виконати зміну, можна скористатися операцією добування зрізу, яка повертає зазначений фрагмент рядка.

Формат операції: `s[<Початок>:<Кінець>:<Крок>]`

Усі параметри тут 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 рядка.

Слід також помітити, що символ з індексом, зазначеним у цьому параметрі, не входить у фрагмент, що повертається.

3. Якщо параметр `<Крок>` не зазначений, то використовується значення 1. Як значення параметрів можна вказати від'ємні значення.

Використання параметрів при добуванні зрізу

1.Одержуємо копію рядка:

Приклад 4.63.

```
s = "123456789"
ns=s[:]#повертається фрагмент від позиції 0 до кінця рядка
print(ns)
```

Результат: `123456789`

2. Виводимо символи у зворотному порядку:

Приклад 4.64.

```
print(s[::-1])# Від'ємне значення в параметрі<Крок>
```

Результат:`987654321`

```
print(s[::-2])
```

Результат:`97531`

3.Замінимо перший символ у рядку:

Приклад 4.65.

```
"print("S"+s[1:]) #  фрагмент від символу 1 до кінця рядка
```

Результат:S23456789

```
print(s[:-1]+"F")
```

Результат:12345678F

4.Вилучимо останній символ:

Приклад 4.65.

```
>>> s[:-1] # повертається фрагмент від 0 до len(s)-1
'12345678'
```

5.Одержимо перший символ у рядку:

Приклад 4.66.

```
>>> s[:1] #Символ з індексом 1 не входить у діапазон
'1'
```

6.Одержимо останній символ:

Приклад 4.67.

```
>>> s[-1:] #фрагмент від len(s)-1 до кінця рядка
'9'
```

7. Виведемо символи з індексами 2, 3 і 4:

Приклад 4.68.

```
>>> s[2:5] #повертаються символи з індексами 2, 3 и 4
'345'
```

Застосування функції **len()**

Визначити кількість символів у рядку дозволяє функція **len():**

Приклад 4.69.

```
>>> len("Python"),len("\r\n\t"),len(r"\r\n\t")
(6, 3, 6)
```

Тепер, коли ми знаємо кількість символів, можна перебрати всі символи за допомогою циклу **for:**

Приклад 4.70.

```
>>> s = "Python"
>>> for i in range(len(s)):
        print(s[i], end=" ")
```

Результат виконання:

```
P y t h o n
```

Безпосередній перебір елементів рядка

Оскільки рядки підтримують ітерації, ми можемо просто вказати рядок як параметр циклу:

Приклад 4.71.

```
>>> s = "Python"
>>> for i in s:
        print(i, end=" ")
```

Результат виконання буде таким же:

```
P y t h o n
```

З'єднання рядків

1. З'єднати два рядки в один рядок дозволяє оператор +:

Приклад 4.72.

```
>>> print("Рядок1" + "Рядок2")
Рядок1Рядок2
```

2. Крім того, можна виконати неявну конкатенацію рядків. У цьому випадку два рядки вказуються поруч без оператора між ними:

Приклад 4.73.

```
>>> print("Рядок1"  "Рядок2")
Рядок1Рядок2
>>> print("Рядок1"
...     "Рядок2")
...
Рядок1Рядок2
```

Одержання кортежу рядків

Якщо між рядками вказати кому, то ми одержимо кортеж, а не рядок:

Приклад 4.74.

```
>>> s = "Рядок1", "Рядок2"
>>> type(s)  # Одержуємо кортеж, а не рядок
<class 'tuple'>
```

З'єднання змінної й рядка

Якщо з'єднуються змінна й рядок, то слід обов'язково вказувати символ конкатенації рядків, інакше буде виведене повідомлення про помилку:

Приклад 4.75.

```
>>> s = "Рядок1"
>>> print(s + "Рядок2") # Нормально
Рядок1Рядок2
>>> print(s "Рядок2")    # Помилка
Syntaxerror: invalid syntax
```

З'єднання рядка з іншим типом даних

При необхідності з'єднати рядок з іншим типом даних (наприклад, з числом) слід зробити явне перетворення типів за допомогою функції **str ():**

Приклад 4.76.

```
>>> "string" + str(10)
'string10'
```

Інші операції над рядками

Крім розглянутих операцій, рядка підтримують операцію повторення, перевірку на входження й невходження. Повторити рядок зазначена кількість раз можна за допомогою оператора *, виконати перевірку на входження фрагмента в рядок дозволяє оператор **in**, а перевірити на невходження – оператор **not in**:

Приклад 4.77.

```
>>> "-" * 20
'--------------------'
>>> "yt" in "Python"
True
```

```
>>> "yt" in "Perl"
False
>>> "PHP" not in "Python"
True
```

4.3.7. Оператор форматування рядків

Для форматування рядка будемо користуватися такими способами:

1.Перший спосіб: оператор форматування **%**.

<Рядок>="<**%Специфікатори**>**" % <Значення>**

2. Другий спосіб: метод **format().**

<Рядок>="{<Специфікатори>}".format(*args, **kwargs)

3. Додаткові методи: `center, ljust, rjust, zfill`

4. Літеральна інтерполяція рядків **(f - рядки)**

5. Шаблони **(Templates)**

Синтаксис оператора форматування

"<%специфікатори>" % <**Значення**>

Усередині параметра <%специфікатори> можуть бути зазначені специфікатори, що мають наступний синтаксис:

"%[(<Ключ>)][<Прапор>][<Ширина>][.<Точність>]<Тип перетворення>"

1. Кількість специфікаторів усередині рядка повинна дорівнювати кількості елементів у полі <Значення>.

2. Один специфікатор $\Rightarrow$ поле <Значення> може містити одне значення

3. Кілька специфікаторів $\Rightarrow$ перераховуємо значення через кому усередині круглих дужок, створюючи тим самим кортеж.

Приклад 4.78. Як задавати поле <**значення**>

A="**%s**"%10 print(A) *Результат*:10 print("**%s**"%10) *Результат*:10	B="**==%s-%s-%s==**"%(10,20,30) print(B) *Результат*: ==10-20-30== print("**%s-%s-%s**"%(10,20,30)) *Результат*: 10-20-30

Параметр **"%(<Ключ>)"**

(<**Ключ**>) – Ключ словника. Якщо заданий ключ, то в параметрі <**Значення**> необхідно вказати словник, а не кортеж.

Приклад 4.79. Використання ключа словника

```
dat="%(name)s-%(year)s"%{"year":1978,"name":"Nik"}
print(dat)
```

Результат:

```
Nik - 1978
"%(name)s-%(year)s" – рядок спеціального формату
  %(name)s – специфікатор для значення "name":"Nik"
  %(year)s – специфікатор для значення "year":1978
  % – ознака початку специфікатора
s – тип перетворення відповідного значення
(name), (year)- ключі
```

%<Прапор #> та тип перетворення

Прапор #

Застосовуємо разом з типами перетворення: o, x, X, f,F,.f,.F

#o – вісімкові символи 0o,

#x, #X - шістнадцяткові – 0x, 0X

#f,#F для дійсних чисел (6 знаків після крапки з округленням)

#.f, #.F для дійсних чисел (0 знаків після крапки з округленням)

Приклад 4.80.

```
>>> print("%#o  %#o  %#o" % (0o77, 0b10, 12))
0o77 0o2 0o14
>>> print("%#x  %#x %#x" % (0xff, 10, 12))
0xff 0xa 0xc
>>>print("%#X  %#X  %#X" % (0xff, 10, 12))
0XFF 0XA 0XC
>>> print("%#.F  %#.F" % (300.1, 300.9))
300. 301.
```

```
>>> print("%#F    %#F" % (300.12345678, 300.123))
300.123457    300.123000
print("%#.3F    %#.2F" % (300.123, 300.12))
```

%<Прапор 0> та **%<Прапор ->**

0 (нуль) - задає наявність провідних нулів для числового значення:

Приклад 4.81.

```
>>> "%d" % 3
'3'
>>> "%d" % 3,6
'3'
>>> "%d - %05d" % (3,3)# 5 - ширина поля
"3 - 00003"
```

- – задає вирівнювання по лівій границі області. За замовчуванням використовується вирівнювання по правій границі. Якщо прапор зазначений одночасно з прапором 0, то дія прапора 0 буде скасована.

Приклад 4.82.

```
>>> "%5d - %-5d" % (3, 3)#5- ширина поля
'    3 - 3    '
>>> "'%05d' - '%-05d'" % (3, 3)
"'00003' - '3    '"
```

%<Прапор +> та **%<Прапор пробіл>**

пробіл – вставляє пробіл перед додатним числом. Перед від’ємним числом буде стояти мінус.

Приклад 4.83.

```
>>>"% d - % d" % (-3, 3)
'-3 -  3'
```

+ – задає обов'язковий вивід знака як для від’ємних, так і для додатних чисел. Якщо прапор + зазначений одночасно з прапором пробіл, то дія прапора пробіл буде скасована.

Приклад 4.84.

```
>>> "'%+d' - '%+d'" % (-3, 3)
"'-3' - '+3'"
```

%<Ширина> мінімальна ширина поля

Якщо рядок не вміщується в зазначену ширину, то значення ігнорується, і рядок виводиться повністю:

Приклад 4.85.

```
>>> '%10d - %-10d' % (3, 3)
'         3 - 3         '
>>> '%3s%10s' % ("string", "string")
'string    string'
```

Замість значення можна вказати символ "*". У цьому випадку значення слід задати всередині кортежу:

Приклад 4.86.

```
>>> "'%*s''%10s'" % (10, "string", "str")
"'     string''       str'"
```

%<.Точність> кількість знаків після точки для дійсних чисел

Перед цим параметром повинна стояти крапка. Без крапки одержуємо 6 знаків після крапки.

Приклад 4.87.

```
>>> import math
>>>"%s %f %.2f" % (math.pi,math.pi,math.pi)
'3.141592653589793 3.141593 3.14'
```

Замість значення можна вказати символ «*». У цьому випадку значення слід задати всередині кортежу:

Приклад 4.88.

```
>>> "%*.*f" % (8, 5, math.pi)8-загальна довжина
' 3.14159'
>>>a=3
>>>'%.*f' % (a, math.pi)
```

'3.142'

%<Тип перетворення>

Параметр є обов'язковим!

У параметрі <**Тип перетворення**> можуть бути зазначені наступні символи: **s,r,a,c,d,i,o,x,X,f,F,e,E,g,G**

Кожний із символів указує на виконання відповідних дій по перетворенню:

s – використовує функцію `str():print("%s"%3)` ⟹ `3`

r – використовує функцію `repr():print("%r"%3)` ⟹ `3`

a – використовує функцію:ascii(): print("%a"%'ф')⟹ '\u0444'

c – перетворює код символа в символ: `print("%c"%104)` ⟹ `h`

d i- повертають цілу частину числа: `print("%d"%11.6)` ⟹ `11`

o – вісімкове значення: print `("%o"%8)` ⟹ `10`

x,X – шістнадцяткове значення: `print("%x"%15)` ⟹ `f`

f,F – дійсне число в десятковій формі: print("%f"%3) ⟹ 3.000000

e,E – дійсне число в експонентній формі: print("%e"%33.1) ⟹ 3.310000e+01

g,G – Еквівалентно `f,F` і `e,E`

Символи s і r у параметрі <Тип перетворення>

s – перетворює будь-який об'єкт у рядок за допомогою функції `str()`:

Приклад 4.89

```
>>> print("%s" % ("Звичайний рядок"))
Звичайний рядок
>>>print("%s %s %s" % (10, 10.52, [1, 2, 3]))
10 10.52 [1, 2, 3]
```

r – перетворює будь-який об'єкт у рядок за допомогою функції `repr()`.

Приклад 4.90.

```
>>> print ("%r" % ("Звичайний рядок"))
'Звичайний рядок'
```

Символи a і c у параметрі <Тип перетворення>

a – перетворює об'єкт у рядок за допомогою функції `ascii()`. У випадку

кирилиці:

Приклад 4.91.

```
>>> print("%a" % ("рядок"))
'\u0440\u044f\u0434\u043e\u043a'
```

c – виводить одиночний символ або перетворює числове значення на символ. Як приклад виведемо числове значення й відповідний до цього значення символ:

Приклад 4.92.

```
>>> for i in range(33, 127): print("%s => %c" % (i, i))
65 => A
>>>print("%a => %c" % ('p', '\u0440'))
'\u0440' => p
```

Символи d, i і o у параметрі <Тип перетворення>

d i **i** – повертають цілу частину числа:

Приклад 4.93.

```
>>> print("%d %d %d" % ( 10, 25.6, -80))
10 25 -80
>>> print("%i %i %i" % ( 10, 25.6, -80))
10 25 -80
```

o – вісімкове значення:

Приклад 4.94

```
>>> print("%o-%o-%o"%(0o77,10,3))
77-12-3
>>> print("%#o-%#o-%#o"%(0o77,10,3))
0o77-0o12-0o3
```

Символи x і X у параметрі <Тип перетворення>

x – шістнадцяткове значення в нижньому регістрі:

Приклад 4.95

```
>>> print("%x %x %x" % (0xff, 10, 5))
ff a 5
>>> print("%#x %#x %#x" % (0xff, 10, 5))
```

```
0xff 0xa 0x5
```

X – шістнадцяткове значення у верхньому регістрі:

```
>>> print("%X %X %X" % (0xff, 10, 5))
FF A 5
>>> print("%#X %#X %#X" % (0xff, 10, 5))
0XFF 0XA 0X5
```

Символи f і F у параметрі **<**Тип перетворення**>**

f і **F** – дійсне число в десятковій формі:

```
>>> print("%f %f %f" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
>>> print("%F %F %F" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
>>> print("%#.0F %#.0F" % (300, 300))
300. 300.
print("%.F %#.0F %#.F" % (300, 300, 300))
300 300. 300.
print("%.2F %#.2F" % (300, 300))
300.00 300.00
```

Символи e і E у параметрі **<**Тип перетворення**>**

e – дійсне число в експонентній формі (буква «**e**» у нижньому регістрі):

Приклад 4.96.

```
>>> print("%e %e" % (3000, 18657.81452))
3.000000e+03 1.865781e+04
```

E – дійсне число в експонентній формі (буква «**E**» у верхньому регістрі):

Приклад 4.97.

```
>>> print("%E %E" % (3000, 18657.81452))
3.000000E+03 1.865781E+04
```

Символи g і G у параметрі **<**Тип перетворення**>**

g – еквівалентно **f** або **e** (вибирається більш короткий запис числа):

Приклад 4.98.

```
>>> print("%g %g %g" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578e-05 1.865e-05
```

G - еквівалентно **F** або **E** (вибирається більш короткий запис числа):

```
>>>print("%G %G %G" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578E-05 1.865E-05
```

Вивід службових символів

Якщо усередині рядка необхідно використовувати символ відсотка (%), то цей символ слід подвоїти (%%), інакше буде виведене повідомлення про помилку:

Приклад 4.99.

```
>>> print("% %s" % ("- це символ відсотка"))
# Помилка
Traceback (most recent call last):
File "<Input>", line 1, in <module>
print("% %s" % ("- це символ відсотка")) # Помилка
Typeerror:not all arguments converted during string
formatting
# Нормально
>>> print("%% %s" % ("- це символ відсотка"))
% - це символ відсотка
```

Досягти такого ж виводу можна так

```
>>> print("%s" % ("% - це символ відсотка"))
```

4.3.8 Методи форматування

```
center ( <Ширина> [, <Символ> ])
```

Здійснює вирівнювання рядка по центру усередині поля зазначеної ширини. Якщо другий параметр не зазначений, то справа і зліва від початкового рядка будуть додані пробіли.

Приклад 4.100.

```
>>> s = "str"
>>> s.center(15), s.center(11, "-")
```

```
('    str    ', '----str----')
```

Приклад спільного використання методу center і прапора - для вирівнювання

Виконаємо вирівнювання трьох фрагментів шириною 5 символів. Перший фрагмент буде вирівняний по правому краю, другій – по лівому, а третій – по центру:

Приклад 4.101.

```
>>> s = "str"
>>> "'%5s' '%-5s' '%s'" % (s,s,s.center(5))
"'  str' 'str  ' ' str '"
```

Якщо кількість символів у рядку перевищує ширину поля, то значення ширини ігнорується, і рядок повертається повністю:

Приклад 4.102.

```
>>> s = "string"
>>> s.center(6), s.center(5)
('string', 'string')
```

Метод форматування `ljust ( <Ширина> [, <Символ>] )`

Виконує вирівнювання рядка по лівому краю усередині поля зазначеної ширини. Якщо другий параметр не зазначений, то праворуч від початкового рядка будуть додані пробіли. Якщо кількість символів у рядку перевищує ширину поля, то значення ширини ігнорується, і рядок повертається повністю.

Приклад 4.103.

```
>>> s = "string"
>>> s.ljust(15), s.ljust(15, "-")
('string         ', 'string---------')
>>> s.ljust(6), s.ljust(5)
('string', 'string')
```

Метод форматування `rjust(<Ширина>[, <Символ>])`

Виконує вирівнювання рядка по правому краю усередині поля зазначеної ширини. Якщо другий параметр не зазначений, то ліворуч від початкового

рядка будуть додані пробіли. Якщо кількість символів у рядку перевищує ширину поля, то значення ширини ігнорується, і рядок повертається повністю.

Приклад 4.104.

```
>>> s = "string"
>>> s.rjust(15), s.rjust(15, "-")
('           string', '---------string')
>>> s.rjust(6), s.rjust(5)
('string', 'string')
```

Метод форматування `zfill(<Ширина>)`

Виконує вирівнювання фрагмента по правому краю усередині поля зазначеної ширини. Ліворуч від фрагмента будуть додані нулі. Якщо кількість символів у рядку перевищує ширину поля, то значення ширини ігнорується, і рядок повертається повністю.

Приклад 4.105.

```
>>> "5".zfill(20), "123456".zfill(5)
('00000000000000000005', '123456')

>>> s="Рядок"
>>> s.zfill(12), s.zfill(5)
('000000Рядок', 'Рядок')
```

Метод **`format()`**

Крім оператора форматування, ми можемо використовувати для цієї ж мети метод `format()`. Він має наступний синтаксис:

`<Рядок>="{<спец.форм>} {<спец.форм>}".format(s, *m, **n)`

У параметрі `<Рядок спец.формату>` усередині фігурних дужок: `{ }` – вказуються специфікатори, що мають наступний синтаксис:

`{[<Поле>][! <Функція>][ : <Формат>]}`

1. Усі символи, розташовані поза фігурними дужками, виводяться без перетворень.

2. Якщо всередині рядка необхідно використовувати фігурні дужки { }, то ці символи слід подвоїти, інакше виконується виключення `ValueError`.

Приклад 4.106.

```
>>> print("{0}".format("Мінімальний набір параметрів"))
Мінімальний набір параметрів
>>> print("Символи {{ та }} -
{0}".format("спеціальні"))
Символи { та } - спеціальні
```

Використання параметра <Поле>

1.Можна вказати індекс позиції (нумерація починається з нуля) або ключ.

2.Дозволено комбінувати позиційні й іменовані параметри. У цьому випадку в методі `format()` іменовані параметри вказуються в самому кінці.

Приклад 4.107. Указуємо індекс для параметрів:

```
>>> "{0} - {1} - {2}".format(10,12.3, "string")# Індекси
'10 - 12.3 - string'
>>> "{1} - {0} - {2}".format(10,12.3, "string")# Індекси
'12.3 - 10 - string'
```

Указуємо список у параметрі *m:

```
>>> m = [10, 12.3, "string"]
>>> "{2} - {1} - {0}".format(*m)# Індекси
'string - 12.3 - 10'
```

Менше індексів ніж елементів – можливо, більше - помилка

Робота з ключом елемента словника та іменованими параметрами

Іменовані параметри:

```
>>>"{model}-{color}".format(color="red", model ="BMW")#ключі
'BMW - red'
```

Указуємо словник у параметрі **n:

```
>>> d = {"color": "red", "model": "BMW"}
>>> "{model}-{color}".format(**d) # Ключі
'BMW-red'
```

Змішана вказівка: іменовані параметри в кінці списку

```
>>>"{color} - {0}".format(2015, color="red") #
Комбінація
'red - 2015'
```

Об'єкт як параметр методу **format()**

1. Як параметр в методі **format()** можна вказати послідовність.

2. Для доступу до елементів по індексу всередині рядка формату застосовуються квадратні дужки.

Приклад 4.108.

```
>>> arr = [10, [12.3, "string"]]
>>> "{0[0]} – {0[1][0]} - {0[1][1]}".format(arr)
#{0[0]} – об'єкт 0, елемент 0
#{0[1][0]} – об'єкт 0, елемент 1, елемент 0
#{0[1][1]} – об'єкт 0, елемент 1, елемент 1
'10 - 12.3 - string'
# Індекси
>>>"{a[0]} - {a[1][1]}".format(a=arr)
'10 - string'
```

Коротка форма запису без параметра **<Поле>**

Існує також коротка форма запису, при якій параметр <Поле> не вказується. У цьому випадку дужки без зазначеного індексу нумеруються зліва направо, починаючи з нуля:

Приклад 4.109.

```
>>>"{} - {} - {} - {n}".format ( 1, 2, 3, n=4)
# "{0} – {1} – {2} – {n}"
'1 – 2 – 3 – 4'
>>>"{} - {} - {n} - {}".format(1, 2, 3, n= 4)
# "{0} – {1} – {n} – {2}"
'1 – 2 – 4 – 3'
```

Використання параметра <Функція>

Задає функцію, за допомогою якої обробляються дані перед вставкою в рядок. (аналогічно до типу перетворення в операторі %)

`s` – дані обробляються функцією `str ()`,

r - функцією `repr()`,

`a` – функцією `ascii ()`.

Якщо параметр не зазначений, то для перетворення даних у рядок використовується функція `str ()`.

Приклад 4.110.

```
>>> print("{0!s}".format("рядок"))#str()рядок
>>> print("{0!r}".format("рядок"))#repr()'рядок'
>>> print("{0!a}".format("рядок"))  # ascii()
' '\u0440\u044f\u0434\u043e\u043a'
```

Використання параметра <Формат>

{[<Поле>][! <Функція>][: <Формат>]}

У параметрі <Формат> вказується значення, що має наступний синтаксис:

[[<Заповнювач>] <Вирівнювання>] [<Знак>] [#] [0] [<Ширина>][,][.<Точність>][<Перетворення>]

Підпараметр <Ширина>

Задає мінімальну ширину поля виводу. Якщо рядок не міститься в зазначеній ширині, то значення ігнорується, і рядок виводиться повністю:

Приклад 4.111.

```
>>>"'{0:10}' '{1:3}'".format(3, "string")
"'         3' 'string'"
```

Передача підпараметра <Ширина>

Ширину поля можна передати як параметр в методі format(). У цьому випадку замість числа вказується індекс параметра всередині фігурних дужок:

Приклад 4.112

```
for i in range(1,5):
 print("'{0:{1}}'".format(3,i))
```

Результат:

```
'3'
' 3'
'  3'
'   3'
```

За замовчуванням значення всередині поля вирівнюється по правому краю.

Підпараметр <Вирівнювання>

Управляє вирівнюванням. Можна вказати наступні значення: **<, >,^,=**

< – по лівому краю;

> – по правому краю;

^ – по центру поля.

Приклад 4.113.

```
>>>"'{0:<10}' '{1:>10}' '{2:^10}'".format(3, 3, 3)
"'3         ' '         3' '     3      '"
```

= – знак числа вирівнюється по лівому краю, а число по правому краю

Приклад 4.114.

```
>>>"'{0:=10}' '{1:=10}'".format(-3, 3)
"'-        3' '         3'"
```

<Вирівнювання**>** з заповненням нулями

1. Як видно з наведеного прикладу, простір між знаком і числом за замовчуванням заповнюється пробілами.

2. Знак додатного числа не вказується.

3. Щоб замість пробілів простір заповнювався нулями, необхідно вказати нуль перед шириною поля

Приклад 4.115.

```
>>> "'{0:=010}' '{1:=010}'".format(-3, 3)
"'-000000003' '0000000003'"
>>> "'{0:=05}' '{1:=06}'".format(-123, 34)
"'-0123' '000034'"
```

Підпараметр <Заповнювач>

1. Такого ж ефекту можна досягтися, указавши нуль у підпараметрі <Заповнювач>.

2. У цьому підпараметрі допускаються інші символи, які будуть виводитися замість пробілів:

Приклад 4.116.

```
>>>"'{0:0=10}' '{1:0=10}'".format(-3, 3)
"'-000000003' '0000000003'"
>>>"'{0:*<10}' '{1:+>10}' '{2:10}'".format(3, 3, 3)
"'3********* '+++++++++3' '         3'"
```

Параметр **<Знак>.**

Припустимі значення цього параметра:

<+, пробіл, відсутність параметрів>

+ – задає обов'язковий вивід знака як для від’ємних, так і для додатних чисел;

параметри відсутні – вивід знака тільки для від’ємних чисел (значення за замовчуванням);

пробіл – вставляє перед додатним числом. Перед від’ємним числом буде стояти мінус.

Приклад 4.117.

```
>>>"'{0:+}' '{1:+}' '{0:-}' '{1:-}'".format(3,-3)
"'+3' '-3' '3' '-3'"
>>>"'{0: }' '{1: }'".format(3,-3)# Пробіл
"' 3' '-3'"
```

Параметр <**Перетворення**>

Для цілих чисел у параметрі <**Перетворення**> можуть бути зазначені наступні опції: **b,c,d,n,o,x,f,e,g,X,F,E,G**

b – двійкове значення:

Приклад 4.118.

```
>>>"'{0:b}' '{0:#b}'".format(3)
"'11' '0b11'"
```

c – перетворює ціле число у відповідний символ:

Приклад 4.119.

```
>>>"{0:c}".format(100)
'd'
```

d – десяткове значення;

n – аналогічно опції **d**, але враховує настроювання локалі. Наприклад, виведемо велике число з поділом тисячних розрядів комами:

Приклад 4.120

```
>>> print("{0:,d}".format(1000000000))
1,000,000,000
>>> print('{0:,d}'.format(1000000000).replace(',', '
'))
1 000 000 000 # пропуски між розрядами
```

o – вісімкове значення:

```
>>> "'{0:d}' '{0: o}' '{0:#o}'".format(511)
"'511' '777' ' 0o777'"
```

x – шістнадцяткове значення в нижньому регістрі:

```
>>> "'{0:x}' '{0:#x}'".format(255)
"'ff ' '0xff'"
```

X – шістнадцяткове значення у верхньому регістрі:

```
>>> "'{0:X}' ' {0:#X}'".format(255)
"'FF' '0XFF'"
```

Підпараметр <**Перетворення**>

Для дійсних чисел у параметрі <**Перетворення**> можуть бути зазначені наступні опції: **f,F**

f і **F** – дійсне число в десятковій формі:

Приклад 4.121.

```
>>>"'{0:f}' '{1:f}' '{2:f}'".format(30, 18.6578145, -2.5)
"'30.000000' ' 18.657815' '-2.500000'"
```

Підпараметр <**.Точність**>

За замовчуванням виведене число має шість знаків після коми. Задати інше кількість знаків після коми можна в параметрі <**Точність**>:

Приклад 4.122.

```
print("{0:10f} {1:.2f}".format(18.6578145, -2.5))
```

```
.18.657815' '-2.50'"
```

`{0:10f}`- виділяє 10 місць під дійсне число

`{0:.2f}`- виводить 2 знаки після крапки

```
print("{0:2f} {1:.2f}".format(18.6578145, -2.556))
```

18.657815 -2.56

`{0:2f}`- якщо 2 –це менше за довжину числа, то залишається попередня довжина

`{0:.2f}`- якщо .2 менше мантиси, то округлення

e – дійсне число в експонентній формі (буква e в нижньому регістрі):

Приклад 4.123.

```
>>>"'{0:e}' '{1:e} '".format(3000, 18657.81452)
"'3.000000e+03' '1.865781e+04'"
```

E – дійсне число в експонентній формі (буква E в верхньому регістрі):

Приклад 4.124.

```
>>> "'{0:E}' '{1:E}'".format(3000, 18657.81452)
"'3.000000E+03' '1.865781E+04'"
```

Тут за замовчуванням кількість знаків після коми також рівно шести, але ми можемо вказати іншу величину цього параметра:

Приклад 4.125.

```
>>> "'{0:.2e}' '{1:.2E}'".format(3000, 18657.81452)
"'3.00e+03' '1.87E+04'"
```

g - еквівалентно **f** або **e** (вибирається більш короткий запис числа):

Приклад 4.126.

```
>>>"'{0:g}' '{1:g}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578e-05'"
```

n - аналогічно опції **g**, але враховує настроювання локалі;

G - еквівалентно **f** або **E** (вибирається більш короткий запис числа):

Приклад 4.127.

```
>>>"'{0:G}' '{1:G}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578E-05'"
```

% – множить число на 100 і додає символ відсотка в кінець. Значення відображається відповідно до опції **f**.

Приклад 4.128.

```
>>>"'{0:%}' '{1:.4%}'".format(0.086578, 0.000086578)
"'8.657800%' '0.0087%'"
```

Літеральна інтерполяція рядків (f-рядки)

Цей підхід не виключає використання розглянутих методів форматування. f-рядок – це літеральний рядок з префіксом «f», який містить вирази у середині фігурних дужок. При виконанні вирази замінюються значеннями.

Приклад 4.129.

```
import datetime
name = 'Петро'
last="Петренко"
age = 18
birthday = datetime.date(2002, 10, 12)
print(f'Мене звуть {name}. Мій вік - {age} років.')
print(f'Я народився {birthday:%A, %B %d, %Y}.')
print(f'Моє прізвище {last!r}.')
```

Результат

```
Мене звуть Петро. Мій вік - 18 років.
Я народився Saturday, October 12, 2002.
Моє прізвище 'Петренко'.
```

Шаблонізація (Templating)

Модуль string включає універсальний клас Template із спрощеним синтаксисом, придатним для редагування кінцевими користувачами. Це дозволяє користувачам налаштовувати свої програми, не змінюючи їх.

Формат використовує імена заповнювачів, які починаються з символап $. Далі йдуть буквено-цифрові символи та підкреслення.

```
>>> from string import Template
>>> t = Template('Привіт, $name!')
>>> t.substitute(name='Іван')
'Привіт, Іван!'
>>> t.substitute(name='Петро')
'Привіт, Петро!'
```

4.3.9. Функції для роботи з рядками

str ([<Об'єкт>[,<Кодування>[,<Обробка помилок>]]])

<Обробка помилок>:**"strict","replace","ignore"**

– перетворює будь-який об'єкт у рядок. Якщо параметр не зазначений, то повертається порожній рядок. Використовується функцією print() для виводу об'єктів.

Приклад 4.130.

```
>>>str()
''
>>>str([1,2])
'[1, 2]'
>>>str((3,4))
'(3, 4)'
>>>str({"x": 1})
"{'x': 1}"
```

>>>str("рядок1\nрядок2") 'рядок1\nрядок2'	>>>print(str("рядок1\nрядок2")) рядок1 рядок2

repr(<Об'єкт>) – повертає строкове представлення об'єкта.

Приклад 4.131.

```
>>>repr("Рядок")
"'Рядок'"
>>>repr([1,2,3])
'[1, 2, 3]'
>>>repr({"x": 5})
"{'x': 5}"
```

Інтерпретатор:

```
>>> repr("рядок1\nрядок2")
"'рядок1\\nрядок2'"
```

Компілятор:

```
print(repr("рядок1\nрядок2"))
'рядок1\nрядок2'
print(repr("рядок1\nрядок2\"))
SyntaxError: EOL while scanning string literal
```

ascii(<Об'єкт>)- повертає строкове представлення об'єкта. У рядку можуть бути символи тільки з кодування ASCII.

Приклад 4.132.

```
>>> ascii([1, 2, 3]), ascii({"x":5})
('[1, 2, 3]', "{'x': 5}")
>>> ascii("рядок")
"'\\u0440\\u044f\\u0434\\u043e\\u043a'"
```

Порівняйте

```
print(str("рядок1\nрядок2"))
print(repr("рядок1\nрядок2"))
print(ascii("рядок1\nрядок2"))
рядок1
рядок2
'рядок1\nрядок2'
```

'\u0440\u044f\u0434\u043e\u043a1\n\u0440\u044f\u0434\u043e\u043a2'

`len(<Рядок>)` – повертає кількість символів у рядку:

Приклад 4.133

```
>>>len("Python"), len("\r\n\t"),len(r"\r\n\t")
(6, 3, 6)
>>> len("рядок") #Рахуємо символи
5
>>> a=bytes("рядок", "utf-8")
>>> len(a) #Рахуємо байти
10
>>> a=bytearray("рядок", "utf-8")
>>> len(a) #Рахуємо байти
10
```

4.3.10. Методи для роботи з рядками

`strip([<Символи>])` – видаляє зазначені в параметрі символи на початку й наприкінці рядка. Якщо параметр не заданий, видаляються «пропускові» (пробільні) символи: пробіл, символ переводу рядка (`\n`), символ повернення каретки (`\r`), символи горизонтальної (`\t`) і вертикальної (`\v`) табуляції:

Приклад 4.134. Синтаксис методу: s.strip()

```
>>> s1= "\vstr\n\r\t"
>>> s2 ="strstrstrokstrstrstr"
>>> print("%s - %s" % (s1.strip(), s2.strip("tsr")))
str - ok
>>> s1, s2 = "str\nm\r\v\t", "strstrstrokstrsmrstr"
>>>print("%s - %s" % (s1.strip(), s2.strip("tsr")))
str
m – okstrsm
```

`lstrip([<Символи>])` – видаляє пробільні або зазначені символи на початку рядка:

Приклад 4.135.

```
>>> s1, s2 = " \tstr ", "strstrstrokstrstrstr"
>>> print("%s - %s" % (s1.lstrip(), s2.lstrip("tsr")))
str - okstrstrstr
```

`rstrip([<Символи>])` – видаляє пробільні або зазначені символи наприкінці рядка:

Приклад 4.136.

```
>>> s1, s2 = " str\t ", "strstrstrokstrstrstr"
>>> print("%s -%s" % (s1.rstrip(), s2.rstrip("tsr")))
 str - strstrstrok
```

`split ([<Роздільник>[,<Ліміт>]])` – розділяє рядок на підрядки по зазначеному роздільнику й додає ці підрядки в **список**, який повертається як результат.

1. Якщо `<Роздільник>` не зазначений або має значення None, то як роздільник використовується символ пробілу.
2. В`<Ліміт>` можна задати кількість підрядків у результуючому списку.
3. Якщо `<Ліміт>` не зазначений або дорівнює -1, у список потраплять усі підрядки.
4. Якщо підрядків більше зазначеної кількості, то список буде містити ще один елемент – із залишком рядка.

Приклад 4.137. Дія методу `split`

```
>>> s = "word1 word2 word3"
>>> s.split(), #параметри не задано - пробіл
['word1','word2','word3']
>>> s.split(None, -1), #None - пробіл
['word1','word2','word3']
>>> s.split(None, 1) # 1 –одне розбиття
['word1','word2 word3']
>>> s = "word1\nword2\nword3"
```

```
>>> s.split("\n")# \n -роздільник
['word1', 'word2', 'word3']
>>> s = "word1-word2-word3"
>>> s.split("-")#"-"-роздільник
['word1', 'word2', 'word3']
```

Якщо в рядку містяться кілька пробілів підряд і роздільник не зазначений, то порожні елементи не будуть додані в список:

Приклад 4.138.

```
>>> s = "word1      word2          word3  "
>>> s.split()
 ['word1', 'word2', 'word3']
```

При використанні іншого роздільника можуть виникнути порожні елементи:

```
s = ",,word1,,word2,,word3,,"
print(s.split(",")) # , - роздільник
['','','word1','','word2','','word3','','']
>>> "1,,2,,3".split(",")
['1', '', '2', '', '3']
```

Якщо роздільник не знайдений у рядку, то список буде складатися з одного елемента, що представляє початковий рядок:

```
>>> "word1 word2 word3".split("\n")
['word1 word2 word3']
```

`rsplit([<Роздільник>[,<Ліміт>]])` **-** аналогічний методу `split()`, але пошук символу-роздільника проводиться справа наліво.

Приклад 4.139.

```
>>> s = "word1 word2 word3"
>>> s.rsplit()
['word1', 'word2', 'word3']

>>> s = "word1 word2 word3"
s.rsplit(None,1)
```

```
['word1 word2', 'word3']
>>> "word1\nword2\nword3".rsplit("\n")
['word1', 'word2', 'word3']
```

`splitlines([True])` – розділяє рядок на підрядки по символу переводу рядка `(\n)` і додає їх у список.

1. Символи переводу рядка включаються в результат, тільки якщо необов'язковий параметр має значення `True`.

2. Якщо роздільник не знайдений у рядку, то список буде містити тільки один елемент.

Приклад 4.140.

```
>>> "word1\nword2\nword3".splitlines()
[ 'word1', 'word2', 'word3']
# - True – роздільник включено у список
>>> "word1\nword2\nword3".splitlines(True)
['word1\n', 'word2\n', 'word3']
# - False - еквівалентно пустому параметру
>>> "word1\nword2\nword3".splitlines(False)
[ 'word1', 'word2', 'word3']
>>>"word1 word2 word3".splitlines()#Роздільника немає
['word1 word2 word3']
```

`partition(<Роздільник>)` – знаходить перше входження символу-роздільника в рядок і **повертає кортеж** із трьох елементів:

1. Перший елемент буде містити фрагмент, розташований перед роздільником.

2. Другий елемент - сам роздільник,

3. Третій елемент – фрагмент, розташований після роздільника.

4. Пошук проводиться зліва направо.

5. Якщо символ-роздільник не знайдений, то перший елемент кортежу буде містити весь рядок, а інші елементи залишаться порожніми.

Приклад 4.141. Аргумент обов’язковий!!

```
>>> "word1 word2 word3".partition(" ")
```

```
('word1', ' ', 'word2 word3')
>>> "word1 word2 word3".partition("\n ")
('word1 word2 word3', '', '')
```

rpartition (<Роздільник>) – метод аналогічний методу partition(), але пошук символу-роздільника проводиться справа наліво.

1. Якщо символ-роздільник не знайдений:

- перші два елементи кортежу виявляться порожніми,

- третій елемент буде містити весь рядок.

Приклад 4.142.

```
>>> "word1 word2 word3".rpartition(" ")
( 'word1 word2 ', ' ', 'word3' )
>>> "word1 word2 word3".rpartition("\n" )
('', '', 'word1 word2 word3')
```

join() – перетворює послідовність у рядок. Елементи додаються через зазначений роздільник. Формат методу:

```
<Рядок> = <Роздільник>.join(<Послідовність>)
```

Як приклад перетворимо список і кортеж у рядок:

Приклад 4.143.

```
>>> "=>" .join(["word1", "word2", "word3"] )
'word1=>word2=>word3'
>>> " ".join(("word1", "word2", "word3"))
'word1 word2 word3'
>>> "".join(("word1", "word2", "word3"))
'word1word2word3'
```

Елементи послідовностей повинні бути рядками, інакше виконується виключення TypeError:

Приклад 4.144.

```
>>> " ".join(("word1", "word2", 5))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: sequence item 2: expected str instance, int
found
```

Рядки є незмінюваними типами даних. Якщо спробувати змінити символ по індексу, то виникне помилка.

Технологія зміни символів рядка

1. Щоб змінити символ по індексу, можна перетворити рядок у список за допомогою функції `list()`.

2. Виконати зміни символів

3. За допомогою методу `join()` перетворити список знов у рядок.

Приклад 4.145.

```
s = "Python"
arr = list(s); # Перетворимо рядок у список
print(arr)
arr[0] = "J";
print(arr)
s = "".join(arr)
print(s)
```

Результат:

```
['P', 'y', 't', 'h', 'o', 'n']
['J', 'y', 't', 'h', 'o', 'n']
'Jython'
```

Інший спосіб зміни рядка

1. Перетворити рядок типу `str` у тип `bytearray`,

2. Змінити символ по індексу.

3. Виконати зворотне перетворення

Приклад 4.146.

```
s = "python"
b = bytearray(s, "cp1251")
print(b)
b[0] = ord("j" )
print(b)
s = b.decode("cp1251")
```

```
print(s)
```

Результат:

```
bytearray(b'python')
bytearray(b'jython')
jython
```

Функції для роботи із символами

Для роботи з окремими символами призначені наступні функції:

`Вхідний параметр-код; результат - символ`

`chr(<Код символу>)` – повертає символ по зазначеному коду:

`Вхідний параметр- символ; результат - код`

`ord (<Символ>)` – повертає код зазначеного символу:

Приклад 4.147.

```
>>> print(chr(1055))
П
>>> print (ord ("П"))
1055
```

Зміна регістру символів

Для зміни регістру символів призначені наступні методи:

`upper()` – заміняє всі символи рядка відповідними прописними буквами:

Приклад 4.147.

```
s = "рядок"
print(s.upper())
```

Результат:

```
РЯДОК
```

`lower()` – заміняє всі символи рядка відповідними малими літерами:

Приклад 4.148.

```
s = "рядок"
print(s.lower())
```

Результат:

```
рядок
```

`swapcase()` – заміняє всі малі символи відповідними великими буквами, а всі великі символи – малими:

Приклад 4.149.

```
>>> print("РЯДОК рядок".swapcase())
рядок РЯДОК
```

`capitalize()` – робить першу букву рядка великою:

Приклад 4.150.

```
>>> print("рядок рядок".capitalize())
Рядок рядок
```

`title ()` – робить першу букву кожного слова великою:

Приклад 4.151.

```
>>> s = "перша буква кожного слова стане прописною"
>>> print(s.title())
```

Перша Буква Кожного Слова Стане Прописною

`casefold()` – те ж саме, що й `lower()`, але додатково перетворить усі символи з діакритичними знаками й лігатури в букви стандартної латиниці. Зазвичай застосовується для порівняння рядків:

Приклад 4.152

```
>>> "Python".casefold()=="python".casefold()
True
"der Fluß" casefold()=="der Fluss".casefold()
True
"der Fluß"=="der Fluss"
False
```

Пошук в рядку. Метод **`find()`**

`find()` – шукає підрядок в рядку. Повертає номер позиції, з якої починається входження підрядка в рядок. Якщо підрядок в рядок не входить, то повертається значення -1. Метод залежить від регістру символів.

Формат методу:

<Рядок>.**find**(<Підрядок>[,<Початок>[,<Кінець>]])

Якщо початкова позиція не зазначена, то пошук буде здійснюватися з початку рядка. Якщо параметри <Початок> і <Кінець> зазначені, то проводиться операція добування зрізу:

<Рядок>[<Початок>:<Кінець>]

і пошук підрядка буде виконуватися в цьому фрагменті.

Приклад 4.153.

```
>>> s = "приклад приклад Приклад"
>>> s.find("при")
0
>>> s.find(" при")
7
>>> s.find(" При")
15
>>> s.find("тест")
-1
>>> s.find(" при",9)
-1
>>>s.find(" при",0,6)
-1
>>>s.find(" при",7,12)
7
```

Метод **index()**

index() - метод аналогічний методу find(), але якщо підрядок в рядок не входить, то виконується виключення ValueError.

Формат методу:

<Рядок>.**index**(<Підрядок>[,<Початок>[,<Кінець>]])

Приклад 4.154.

```
>>> s = "приклад приклад Приклад"
>>> s.index(" при")
```

```
7
s.index(" при", 7, 12)
7
s.index(" При", 1)
15
>>> s.index("тест")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: substring not found
```

Метод **rfind()**

rfind() – шукає підрядок в рядку. Повертає позицію останнього входження підрядка в рядок.

Формат методу:

<Рядок>.**rfind**(<Підрядок>[,<Початок>[,<Кінець>]])

- Якщо підрядок в рядок не входить, то повертається значення -1.
- Метод залежить від регістру символів.
- Якщо початкова позиція не зазначена, то пошук буде проводитися з початку рядка.
- Якщо параметри <Початок> і <Кінець> зазначені, то виконується операція добування зрізу, і пошук підрядка буде проводитися в цьому фрагменті.

Приклад 4.155.

```
                0 1  2 3 4  5 6 7  8 9  10 11 12 13 14 15 16 1718 19 20 22 23 24 26 27 28 29 30
>>> s = "приклад приклад Приклад Приклад"

>>> s.rfind("при")
8
s.rfind("При")
24
s.rfind("тест")
```

```
-1
>>> s.find(" при",0,61)
7
s.find(" При",11,20)
15
```

Метод **rindex()**

`rindex()` **-** метод аналогічний методу `rfind()`,

Формат методу**:**

`<Рядок>.`**`rindex`**`(<Підрядок>[,<Початок>[,<Кінець>]])`

Якщо підрядок в рядок не входить, то виконується виключенняя `ValueError`.

Приклад 4.156.

```
>>> s = "приклад приклад Приклад Приклад"
>>> s.rindex(" при")
7
>>> s.rindex(" при",0,11)
7
>>> s.rindex("тест")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
Valueerror: substring not found
```

Метод **count()**

`count()` **-** повертає число входжень підрядка в рядок.

Формат методу:

`<Рядок>.`**`count`**`(<Підрядок>[,<Початок>[,<Кінець>]])`

Якщо підрядок в рядок не входить, то повертається значення 0. Метод залежить від регістру символів.

Приклад 4.157.

```
>>> s = "приклад приклад Приклад Приклад"
>>> s.count(" при")
```

```
1
>>> s.count(" при", 6)
1
>>> s.count(" При")
2
>>> s.count("тест")
0
```

Перевірка типу вмісту рядка

(Букви і цифри)`isalnum()` - повертає `True`, якщо рядок містить тільки букви та(або) цифри, а якщо ні, то - `False`.

Якщо рядок порожній, то вертається значення `False`.

Приклад 4.158.

```
>>> "0123".isalnum()
True
>>> "123abc".isalnum()
True
>>> "рядок".isalnum()
True
>>> "".isalnum()
False
>>> "123 abc".isalnum()
False
>>> "abc, 123.".isalnum()
False
```

(Тільки букви) `isalpha()` - повертає `True`, якщо рядок містить тільки букви, а якщо ні, то - `False`.

Якщо рядок порожній, то повертається значення `False`.

Приклад 4.159.

```
>>> "string".isalpha()
True
>>> "рядок".isalpha()
```

```
True
>>> "".isalpha()
False
>>> "123abc".isalpha()
False
>>> "str str".isalpha()
False
>>> "st,st". isalpha()
False
```

(Тільки цифри) `isdigit()` - повертає `True`, якщо рядок містить тільки цифри, а якщо ні, то- `False`:

Приклад 4.160.

```
>>> "0123".isdigit()
True

>>> "1.3".isdigit()
False
>>> "123abc".isdigit()
False
>>> "abcl23".isdigit()
False
>>>"12F".isdigit()
False
```

(Тільки десяткові символи) `isdecimal()` - повертає `True`, якщо рядок містить тільки десяткові символи, а якщо ні, то- `False`.

1. Десятковими символами є не лише десяткові цифри в кодуванні ASCII, а також десяткові цифри в інших мовах.

Приклад 4.161.

```
>>> "123".isdecimal()
True
>>> "123стр".isdecimal()
```

```
False
>>> "⅔".isdecimal()
False
```

Приклад аналізу рядка з використанням методу split() *та* isdecimal()

```
a = input("Ведіть число\n")
c = a.split(".") # Список
print(c)
if len(c)==2:
   if c[0].isdecimal() and c[1].isdecimal():
        result = float(a)
        print("float:", result)
   else: print("str:", a)
elif len(c)==1:
   if c[0].isdecimal():
        result = int(a)
        print("int:", result)
   else: print("str:", a)
```

(Тільки числові символи) `isnumeric()` – повертає `True`, якщо рядок містить тільки числові символи, а якщо ні, то – `False`.

1. Числовими символами є не тільки десяткові цифри в кодуванні ASCII, але символи римських чисел, дробові числа й ін.

Приклад 4.162.

```
>>> "\u2155".isnumeric() # символ ⅕
True
>>> "\u2155".isdigit ()
False
>>> print("\u2155")
⅕
>>> "⅕".isnumeric()
True
```

```
>>> "⅕".isdigit()
False
```

(Тільки верхній регістр) `isupper()` – повертає `True`, якщо рядок містить букви тільки верхнього регістру, а якщо ні, то – `False`:

Приклад 4.163. Не букви можуть бути присутніми в рядку

```
>>> "STRING".isupper()
True
>>> "РЯДОК".isupper()
True
>>> "".isupper()
False
>>> "String1". isupper()
False
>>> "РЯДОК,123".isupper()
True
>>> "123".isupper()
False
>>> "string".isupper(), "String".isupper()
(False, False)
```

(Тільки нижній регістр) `islower()` – повертає `True`, якщо рядок містить букви тільки нижнього регістру, а якщо ні, то – `False`:

Приклад 4.164.

```
>>> "srting".islower()
True
>>> "рядок".islower()
True
>>> "".islower()
False
>>> "string1".islower()
True
>>> "s tr, 123".islower()
True
```

```
"123".islower()
False
>>> "STRING".islower(), "Рядок".islower()
(False, False)
```

(Все з великої букви) `istitle()` – повертає `True`, якщо всі слова в рядку починаються з великої букви, а якщо ні, то – `False`. Якщо рядок порожній, також повертається `False`.

Приклад 4.165.

```
>>> "Str Str".istitle()
True
>>> "Стр Стр".istitle()
True
>>> "Str Str 123".istitle()
True
>>> "Стр Стр 123".istitle()
True
>>> "Str str".istitle(), "Стр стр".istitle()
(False, False)
>>> "".istitle(), "123".istitle()
(False, False)
```

(Тільки символи для друку) `isprintable()` – повертає `True`, якщо рядок містить тільки символи, що друкуються, а якщо ні, то – `False`. Відзначимо, що пробіл є символом, що друкується.

Приклад 4.166.

```
>>>"123".isprintable()
True
>>> "PHP Python".isprintable()
True
>>> "\n".isprintable()
False
```

(пробільні символи)`isspace()` – повертає `True`, якщо рядок містить тільки «пропускові» символи, а якщо ні, то – `False`:

Приклад 4.167.

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
(False, True, False)
```

(може бути змінною)`isidentifier()` – повертає `True`, якщо рядок є припустимим з погляду Python ім'ям змінної, функції або класу, а якщо ні, то – `False`:

Приклад 4.168.

```
>>> "s".isidentifier()
True
>>> "func".isidentifier()
True
>>> "123func".isidentifier()
False
```

Метод `isidentifier ()` лише перевіряє, чи задовольняє задане ім'я правилам мови. Він не перевіряє, чи збігається це ім'я з ключовим словом Python.

(перевірка на ключове слово)`iskeyword ()`

Для перевірки на ключове слово слід застосовувати функцію `iskeyword ()`, оголошену в модулі `keyword`,

`iskeyword ()` повертає `True`, якщо переданий їй рядок збігається з одним із ключових слів:

Приклад 4.169.

```
>>> import keyword
>>> keyword. iskeyword ("else")
True
>>> keyword.iskeyword("elsewhere")
False
```

Програма додавання довільної кількості цілих чисел, введених користувачем

При введенні рядка замість числа програма виводить повідомлення.

Передбачимо можливість введення від’ємних цілих чисел.

Приклад 4.170. Додавання довільної кількості чисел

```
print("Введіть слово 'stop' для отримання результату")
suma = 0
while True:
    x = input("Введіть число: ")
    if x == "stop":
        break   # вихід з циклу
    if x == "":
        print("Ви не ввели значення!")
        continue
    if x[0] == "-": # Якщо першим символом є мінус
        if not x[1:].isdigit():
       # Якщо фрагмент не складається з цифр
            print("Необхідно ввести число, а не рядок!")
            continue
    else: # Якщо мінуса немає, то перевіряємо весь рядок
        if not x.isdigit():
        #  Якщо рядок не складається з цифр
            print("Необхідно ввести число, а не рядок!")
            continue
    x = int(x) # Перетворюємо рядок на число
    suma += x
print("Сума чисел дорівнює:", suma)
```

Процес введення значень і одержання результату має такий вигляд (значення, введені користувачем, виділені напівжирним шрифтом):

```
Введіть слово 'stop' для одержання результату
Введіть число: 10
Введіть число:
Ви не ввели значення!
Введіть число: str
```

```
Необхідно ввести число, а не рядок!
Введіть число: -5
Введіть число: -str
Необхідно ввести число, а не рядок!
Введіть число: stop
Сума чисел дорівнює: 5
```

4.3.11. Налаштування локалі

1. Необхідно підключити модуль за допомогою виразу:

```
import locale
```

2. Для установки локалі (сукупності локальних налаштувань системи) слугує функція `setlocale()` з модуля `locale`.

3.Функція `setlocale()` має наступний формат:

```
setlocale(<категорія>[,<Локаль>));
```

Параметр `<категорія>` може приймати наступні значення:

Значення параметра **<**`категорія`**>**

`locale. LC_ALL` - установлює локаль для всіх режимів;

`locale.LC_COLLATE` - для порівняння рядків;

`locale. LC_CTYPE` - для переводу символів у нижній або верхній регістр;

`locale.LC_MONETARY` - для відображення грошових одиниць;

`locale. LC_NUMERIC` - для форматування чисел;

`locale.LC_TIME` - для форматування виводу дати й часу.

Одержати поточне значення локалі дозволяє функція

`getlocale([<Категорія>]).`

Як приклад налаштуємо локаль під Windows спочатку на кодування Windows-1251, потім на кодування UTF-8, а потім на кодування за замовчуванням.

Далі виведемо поточне значення локалі для всіх категорій і тільки для `locale.LC_COLLATE.`

Приклад 4.171. Застосування `setlocale() i getlocale()`

```
>>> import locale
```

```
>>> # Для кодування windows-1251
>>> locale.setlocale(locale.LC_ALL, "Ukrainian_Ukraine.1251")
' Ukrainian_Ukraine.1251'
>>># встановлюємо локаль за замовчуванням
>>> locale.setlocale(locale.LC_ALL, "")
'Russian Russia.1251'
>>> #отримуємо поточне значення локалі для всіх категорій
>>> locale.getlocale()
( 'Russian_Russia', '1251')
>>> # Одержуємо поточне значення категорії locale. LC COLLATE
>>> locale.getlocale(locale.LC_COLLATE)
( 'Russian_Russia', '1251')
```

Одержання словника з настроюваннями локалі

Одержати налаштування локалі дозволяє функція `localeconv()`. Функція повертає словник з настроюваннями. Результат виконання функції для локалі Ukrainian_Ukraine.1251 виглядає в такий спосіб:

```
>>> locale.localeconv()
{'decimal_point': ',', 'thousands_sep': '\xa0',
'grouping': [3, 0], 'int_curr_symbol': 'UAH',
'currency_symbol': '?', 'mon_decimal_point': ',',
'mon_thousands_sep': '\xa0', 'mon_grouping': [3, 0],
'positive_sign': '', 'negative_sign': '-',
'int_frac_digits': 2, 'frac_digits': 2, 'p_cs_precedes':
0, 'p_sep_by_space': 0, 'n_cs_precedes': 0,
'n_sep_by_space': 0, 'p_sign_posn': 1, 'n_sign_posn': 1}
```

4.4. Створення `bytes` і `bytearray` і робота з даними типами

Навіщо потрібні типи даних `bytes` і `bytearray`

Тип **`str`** використовується для зберігання текстової інформації.

Але що робити, якщо:

Необхідно обробляти зображення або звук?

Необхідно читати дані, задані в різних системах кодування?

Необхідно мімікрувати під об’єкти інших мов програмування?

У цих випадках ми не можемо застосовувати Unicode-рядок.

Для розв'язування цієї проблеми в Python 3 були введені типи `bytes` і `bytearray`, які дозволяють зберігати послідовність цілих чисел у діапазоні від 0 до 255.

В той же час, кожне таке число може позначати код символу.

Тип даних `bytes` є незмінюваним типом, як і рядки,

тип даних `bytearray` – змінюваний, як і списки.

4.4.1. Способи створення об'єкта типу **`bytes`**

Спосіб 1. За допомогою функції **`bytes()`**

`bytes``([<`**`Рядок`**`>,<`**`Кодування`**`>[,<`**`Обробка помилок`**`>]])`

`Наприклад:` **`bytes("Рядок", "cp1251")`**

1. Якщо параметри не зазначені, то повертається порожній об'єкт.

2. Щоб перетворити рядок в об'єкт типу `bytes`, необхідно передати мінімум два перші параметри.

3. Перші два параметри є обов’язковими. Якщо зазначено тільки перший параметр, то виконується виключення `TypeError`.

Приклад 4.172.

```
>>> bytes()
b''
>>> bytes("рядок", "cp1251")
b'\xf0\xff\xe4\xee\xea'
>>> bytes('string')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: string argument without an encoding
```

Третій параметр: <Обробка помилок>

`"strict"` (при помилці виконується виключення `UnicodeEncodeError` –

значення за замовчуванням),

"replace" (невідомий символ заміняється символом питання)

"ignore" (невідомі символи ігноруються).

Приклад 4.172. Приклади обробки помилок

```
>>> bytes("string\ufffd","cp1251","strict")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  …
UnicodeEncodeError: 'charmap' codec can't encode
character '\ufffd' in position 6: character maps to
<undefined>
>>> bytes("string\ufffd", "cp1251", "replace")
b'string?'
>>> bytes ( "string\ufffd", "cp1251", "ignore")
b'string'
```

Спосіб 2 За допомогою методу рядка

Нехай s – змінна типу str. Тоді для неї можна застосувати метод:

```
s.encode([encoding="utf-8"][, errors="strict"]).
```

Наприклад:

```
>>>a="рядок".encode(encoding="cp1251")
>>>a
b'\xf0\xff\xe4\xee\xea'
```

1. Якщо кодування не зазначене, то рядок перетворюється в послідовність байтів у кодуванні UTF-8.

2. У параметрі `errors` можуть бути зазначені значення `"strict"` (значення за замовчуванням),

`"replace","ignore","xmlcharrefreplace"` або `"backslashreplace"`.

Приклад 4.173. Приклади застосування методу `encode`

```
>>> "рядок".encode() #utf-8
```

```
b'\xd1\x80\xd1\x8f\xd0\xb4\xd0\xbe\xd0\xba'
>>> "рядок".encode(encoding="cp1251")
b'\xf0\xff\xe4\xee\xea'
>>>a="рядок\ufffd".encode(encoding="cp1251",errors="xmlcharrefreplace")
>>>a
b'\xf0\xff\xe4\xee\xea&#65533;'
>>>"рядок\ufffd".encode(encoding="cp1251",errors="backslashreplace")
b'\xf0\xff\xe4\xee\xea\\ufffd'
```

Спосіб 3 Додаванням префікса b

1. Указавши букву b або B перед рядком в апострофах, лапках, потрійних апострофах або потрійних лапках.

2. У рядку можуть бути тільки символи з кодами, що входять у кодування ASCII.

3.Усі інші символи повинні бути представлені спеціальними послідовностями:

Приклад 4.174.

```
>>> b"string", b'string', b"""string""", b'''string'''
(b'string', b'string', b'string', b'string')
>>> b"рядок"
Syntaxerror: bytes can only contain ASCII literal
characters.
>>> b'\xf0\xff\xe4\xee\xea'
b'\xf0\xff\xe4\xee\xea'
```

Спосіб 4. За допомогою функції:

bytes(<Послідовність>),

яка перетворює послідовність цілих чисел від 0 до 255 в об'єкт типу **bytes.**

Якщо число не попадає в діапазон, то виконується виключення `ValueError`.

Приклад 4.175.

```
>>> b = bytes([240, 255, 228, 238, 234])
>>> c = bytes((240, 255, 228, 238, 234))
>>> b
```

```
b'\xf0\xff\xe4\xee\xea'
>>>str(b,'cp1251')
'рядок'
>>>str(c,'cp1251')
'рядок'
```

Спосіб 5. (Задавання нульової послідовності)

За допомогою функції

bytes(<Число>),

яка задає кількість елементів у послідовності.

Кожний елемент буде містити нульовий символ:

Приклад 4.176.

```
>>> bytes (10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
    1    2   3  4   5   6   7   8   9   10
>>> bytes (5)
b'\x00\x00\x00\x00\x00'
    1   2   3   4   5
```

Спосіб 6. За допомогою методу

bytes. fromhex (<Рядок>).

<Рядок> містить шістнадцяткові коди символів без пробілів

Приклад 4.177.

```
>>> b = bytes.fromhex('f0ffe4eeea')
>>> b
b'\xf0\xff\xe4\xee\xea'
>>> str(b, "cp1251")
'рядок'
```

1. Об'єкти типу bytes – це послідовності.

2. Елемент такої послідовності зберігає ціле число від 0 до 255, яке позначає код символу.

3. Об'єкти підтримують такі операції:

- доступ до елемента по індексу,
- одержання зрізу,
- конкатенацію,
- повторення й перевірку на входження.

Приклад 4.178. Приклади операцій з типом **bytes**

```
>>> b = bytes ( "string", "cp1251")
>>> b
b'string'
>>> b[0]                # Доступ по індексу
115
>>> b[1:3]              # Одержання зрізу
b'tr'

>>> b+b"123"            # Конкатенація
b'string123'
>>> b*3                 # Повторення
b'stringstringstring'
>>> 115 in b, b"tr" in b, b"as" in b
(True, True, False)
```

Приклад показав, що

при виводі об'єкта цілком і при добуванні зрізу, проводиться спроба відображення символів.

```
>>> b[1:3]              # Одержання зрізу
b'tr'
```

доступ по індексу повертає ціле число, а не символ.

Якщо перетворити об'єкт у список, то ми одержимо послідовність цілих чисел:

Приклад 4.179.

```
>>># Перетворення в список
>>> list(bytes("string","cp1251"))
[115, 116, 114, 105, 110, 103]
```

```
>>> # Доступ по індексу
>>> b[0]
115
```

Тип **bytes** *є незмінюваним типом*

Це означає, що можна одержати значення по індексу, але змінити його не можна:

Приклад 4.180.

```
>>> b = bytes ("string", "cp1251")
>>> b[0] = 168
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'bytes' object does not support item
assignment
```

1. Більшість рядкових методів з типу **str** підтримують об'єкти типу **bytes**
2. Деякі із цих методів можуть некоректно працювати з кирилицею.
3. У такому випадку використовують тип **str**, а не тип **bytes.**

Рядкові методи, не підтримувані об'єктами типу **bytes**

```
encode(),
isidentifier(), isprintable (),
isnumeric (),   isdecimal (),
format()
```

При використанні методів об’єктів типу bytes слід враховувати, що в параметрах потрібно вказувати об'єкти типу `bytes`, а не `str`:

Приклад 4.181.

```
>>> b= bytes( "string", "cp1251")   #функція
>>> c=b.replace(b"s", b"S")         #метод
>>> c
b'String'
>>> b
b'string' #сам об'єкт не змінився
```

```
>>> c is b
False
```

У виразах не можна змішувати рядки й об'єкти типу `bytes`

Попередньо необхідно явно перетворити об'єкти до одного типу, а лише потім виконувати операцію:

Приклад 4.182

```
>>># Помилка при додаванні об'єктів різних типів
>>> b"string" + "string"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can't concat bytes to str

>>> # Рядок привели до типу bytes
>>> b"string" + "string".encode("ascii")
b'stringstring'
```

Однобайтні та багатобайтні дані в типі **`bytes`**

1. Об'єкт типу `bytes` може містити як однобайтні символи, так і багатобайтні.

2. При використанні багатобайтних символів деякі функції змінюють поведінку – наприклад, функція `len()` поверне кількість байтів, а не символів:

Приклад 4.183.

```
>>> #Робота з типом str
>>> len("рядок")
5
>>> #Робота з типом bytes у кодуванні cp1251
>>> len(bytes("рядок","cp1251"))
5
>>> #Робота з типом bytes у кодуванні utf-8
>>> len(bytes("рядок", "utf-8"))
10
```

Метод **decode()** *для об'єктів типу* **bytes**

Перетворити об'єкт типу `bytes` у рядок дозволяє метод `decode()`. Метод має наступний формат:

```
decode([encoding="utf-8"][,errors="strict"])
```

1. Параметр `encoding` задає кодування символів (за замовчуванням `UTF-8`) в об'єкті `bytes`.

2. Параметр `errors` – задає обробку помилок при перетворенні за допомогою параметрів:

`"strict"` (значення за замовчуванням), `"replace"`, `"ignore"`.

Приклад 4.184.

```
>>> b = bytes("рядок", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('рядок', 'рядок')
```

Перетворення за допомогою функції `str()`

Приклад 4.185.

```
>>> b = bytes("рядок", "cp1251")
>>> str(b, "cp1251")
'рядок'
```

Зміна кодування

1. Щоб змінити кодування даних, слід спочатку перетворити тип `bytes` у рядок.

2. Потім зробити зворотне перетворення, указавши потрібне кодування.

Перетворимо дані з кодування Windows-1251 у кодування KOI8-R, а потім назад.

Приклад 4.186.

```
>>> w = bytes("Рядок", "cp1251") # створили об’єкт
#Перекодовуємо w-"cp1251" в k-"koi8-r"
>>> k = w.decode("cp1251").encode("koi8-r")
>>> k, str(k, "koi8-r")#Данные в кодуванні KOI8-R
(b'\xf2\xd1\xc4\xcf\xcb', 'Рядок')
```

```
#Перекодовуємо k-"koi8-r" в w- "cp1251"
>>> w = k.decode("koi8-r").encode("cp1251")
>>> w, str (w, "cp1251")
(b'\xd0\xff\xe4\xee\xea', 'Рядок')
```

Підсумок по типу bytes

Перетворення в* `bytes`*:

1.**`bytes`**([<Рядок>,<Кодування>[,<Обробка помилок>]])

2."рядок".encode(),**`encode`**([encoding="utf-8"][,
errors="strict"])

3.b'string', додавання b: ~~b"рядок"~~ .

4.bytes([225, 226]) #Перетворення list⟹bytes

  list(b'\xe1\xe2')#Перетворення bytes⟹list

5.bytes (10), Пустий bytes

6.bytes.fromhex('f0ffe4eeea') 16-ве ⟹ bytes

Операції з* `bytes`*:

b[0], b[1:3], b"wdf"+b"123", b*3, "w" in b

***Перетворення в* `str`:**

b.decode(encoding="cp1251"),
str(b, "cp1251")

4.4.2. Створення типу даних `bytearray`

1. Тип даних bytearray є різновидом типу bytes і підтримує ті ж самі методи й операції.

2. На відміну від типу bytes, тип bytearray допускає можливість безпосередньої зміни об'єкта й містить додаткові методи, що дозволяють виконувати ці зміни.

Створити об'єкт типу **`bytearray`** можна такими способами:

Спосіб 1. За допомогою функції

bytearray([<Рядок>,<Кодування>[,<Обробка помилок>]])

Наприклад:

```
>>>a=bytearray("рядок", "cp1251")
>>>a
bytearray(b'\xf0\xff\xe4\xee\xea')
```

1. Якщо параметри не зазначені, то повертається порожній об'єкт.

2. Щоб перетворити рядок в об'єкт типу **bytearray**, необхідно передати мінімум два перші параметри.

3. Якщо рядок зазначений тільки в першому параметрі, то виконується виключення **TypeError**.

Приклад 4.187.

```
>>> bytearray()
bytearray(b'')      #повертає порожній об'єкт
>>> bytearray("рядок", "cp1251")
bytearray(b'\xf0\xff\xe4\xee\xea')
>>> bytearray ("рядок ")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: string argument without an encoding
```

Третій параметр:

"strict" (при помилці виконується виключення Unicodeencodeerror – значення за замовчуванням),

"replace" (невідомий символ заміняється символом питання)

"ignore" (невідомі символи ігноруються).

Приклад 4.188. Третій параметр

```
>>> bytearray("string\ufffd", "cp1251","strict")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  …
UnicodeEncodeError: 'charmap' codec can't encode
character '\ufffd' in position 6: character maps to
<undefined>
```

```
>>> bytearray("string\ufffd","cp1251", "replace")
bytearray(b'string?')
>>> bytearray ("string\ufffd", " cp1251", "ignore")
bytearray(b'string')
```

Спосіб 2. За допомогою функції

`bytearray(<Послідовність>)`

1.Перетворює послідовність цілих чисел від 0 до 255 в об'єкт типу `bytearray`.

2.Якщо число не потрапляє в діапазон, то виконується виключення `ValueError`.

Приклад 4.189.

```
>>> b = bytearray([240, 255, 228, 238, 234])
>>> b
bytearray(b'\xf0\xff\xe4\xee\xea')
>>> str(b, "cp1251")
'рядок'
```

Спосіб 3. За допомогою функції `bytearray(<Число>)`

1. Задає кількість елементів у послідовності.

2. Кожний елемент буде містити нульовий символ:

Приклад 4.190.

```
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
>>> bytearray(10)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

Спосіб 4. За допомогою методу `bytearray.fromhex(<Рядок>)`. Рядок у цьому випадку повинен містити шістнадцяткове значення символів:

Приклад 4.191.

```
>>> c = bytearray.fromhex('f0ffe4eeea')
>>> c
bytearray(b'\xf0\xff\xe4\xee\xea')
```

```
>>> str(c, "cp1251")
'рядок'
```

1. Тип `bytearray` є змінюваним типом.

2. Можна не тільки одержати значення по індексу, але й змінити його (що не притаманно рядкам):

Приклад 4.192. Зміна значення по індексу у **`bytearray()`**

```
>>> c = bytearray ("Python", "ascii")
>>> c[0]
80 # Можемо одержати значення
>>> c[0]=b"J"[0] # Можемо змінити значення
>>> c
bytearray(b'Jython')
```

1. Значення, що присвоюється, повинно бути цілим числом у діапазоні від 0 до 255.

2. Щоб одержати число в попередньому прикладі, ми

- створили об'єкт типу `bytearray`,
- присвоїли значення, розташоване по індексу 0 `(c[0]= b"J"[0])`.

Можна, звичайно, відразу вказати код символу, але ж тримати всі коди символів у пам'яті властиво комп'ютеру, а не людині.

4.4.3. Методи модифікації типу `bytearray`

(Числовий код додаємо в кінець)`append (<Число>)` – додає один елемент у кінець об'єкта. Метод змінює поточний об'єкт і нічого не повертає.

Приклад 4.193.

```
>>> c = bytearray("string", "ascii")
>>> c.append(b"1"[0]); c
bytearray(b'string1')
```

(Послідовність додаємо в кінець)`extend(<Послідовність>)` - додає елементи послідовності в кінець об'єкта.

Метод змінює поточний об'єкт і нічого не повертає.

Приклад 4.194.

```
>>> c = bytearray ( "string", "ascii")
>>> c.extend(b"123"); c
bytearray(b'string123')
```

(Послідовність додаємо в кінець)**+ і +=** використовувати оператори для додавання декількох елементів:

Приклад 4.195.

```
>>> c = bytearray("string", "ascii")
>>> c + b"123" # Повертає новий об'єкт
bytearray (b'string123')
>>> c += b"456"; c
bytearray(b'string456') # Змінює поточний об'єкт
```

(Послідовність додаємо в кінець) Присвоювання значення зрізу:

Приклад 4.196.

```
>>> c = bytearray( "string", "ascii")
>>> c[len(c):] = b"123"
# Додаємо елементи в послідовність
>>> c
bytearray(b'string123')
```

(Вставка числового коду в позицію)

`insert(<Індекс>, <Число>)` – додає один елемент у зазначену позицію. Інші елементи зміщаються. Метод змінює поточний об'єкт і нічого не повертає. Додамо елемент у початок об'єкта:

Приклад 4.197.

```
>>> c = bytearray("string", "ascii")
>>> c.insert(0, b"1"[0]); c
bytearray(b'1string')
```

Метод `insert()` дозволяє додати тільки один елемент. Щоб додати кілька елементів, можна скористатися операцією присвоювання значення зрізу. Додамо кілька елементів у початок об'єкта по зрізу:

(Вставка послідовності по зрізу)

```
>>> c = bytearray("string", "ascii")
>>> c[:0] = b"123"; c
bytearray(b'123string')
```

4.4.4. Видалення типу даних **bytearray**

(Видалення елемента по індексу)

`pop([<Індекс>])` – видаляє елемент, розташований по зазначеному індексу, і повертає його.

Якщо індекс не зазначений, то видаляє й повертає останній елемент.

Приклад 4.198.

```
>>> c = bytearray ( "string", "ascii")
>>> c.pop() # Видаляємо останній елемент
103
>>> c
bytearray(b'strin')
>>> c.pop(0)  # Видаляємо перший елемент
115
>>> c
bytearray(b'trin')
```

(Видалення елемента, вибраного по індексу або зрізу)

Вилучити елемент послідовності дозволяє також оператор `del`:

Приклад 4.199.

```
>>> c = bytearray("string", "ascii")
>>> del c[5] # Видаляємо останній елемент
>>> c
bytearray(b'strin')
>>> del c[:2] # Видаляємо перший і другий елементи
>>> c
bytearray(b'rin')
```

(Видалення елемента по числовому коду)

`remove(<Число>)` – видаляє перший елемент, що містить зазначене значення.

1. Якщо елемент не знайдений, виконується виключення `ValueError`.

2. Метод змінює поточний об'єкт і нічого не повертає.

Приклад 4.200.

```
>>> c = bytearray("strstr", "ascii")
>>> c.remove(b"s"[0]) # Видаляє тільки перший елемент
>>> c
bytearray(b'trstr')
```

(Зміна порядку елементів)

`reverse ()` – змінює порядок проходження елементів на протилежний.

1. Метод змінює поточний об'єкт і нічого не повертає.

Приклад 4.201

```
>>> c = bytearray("string", "ascii")
>>> c.reverse()
>>> c
bytearray(b'gnirts')
>>> c = bytearray("123456789", "cp1251")
>>> c.reverse();c
bytearray(b'987654321')
>>> c = bytearray("абвгдеж", "utf-8")
>>> c
bytearray(b'\xd0\xb0\xd0\xb1\xd0\xb2\xd0\xb3\xd0\xb4\xd0\xb5\xd0\xb6')
>>> c.reverse(); c
bytearray(b'\xb6\xd0\xb5\xd0\xb4\xd0\xb3\xd0\xb2\xd0\xb1\xd0\xb0\xd0')
```

4.4.5. Методи і функції перетворення і зберігання

Перетворити об'єкт типу `bytearray` у рядок дозволяє метод `decode()`. Метод має наступний формат:

decode ([encoding="utf-8"] [, errors="strict"])

Наприклад: c.decode(encoding="cp1251")

1. Параметр **encoding** задає кодування символів (за замовчуванням UTF-8) в об'єкті **bytearray**,

2. Параметр **errors –** спосіб обробки помилок шляхом задавання значень:

"strict" (значення за замовчуванням), вивід виключення

"replace"- знаки питання в місцях помилок

"ignore" – ігнорування помилок

Приклад 4.202. Приклад перетворення bytearray в str:

```
>>> c = bytearray("рядок", "cp1251")
>>> c.decode(encoding="cp1251"), c.decode("cp1251")
('рядок','рядок')
>>> c = bytearray("123\n456", "cp1251")
>>> c.decode(encoding="cp1251", errors="replace")
'123\n456'
```

Для перетворення можна також скористатися функцією str():

```
>>> c = bytearray("рядок", "cp1251")
>>> str (c, "cp1251")
'рядок'
```

Підсумок по модифікації **bytearray**

Додавання елементів

```
c.append(b"1"[0]) append (<Число>)
c.extend(b"123")  extend(<Послідовність>)
c += b"456"       + і +=
c.insert(0, b"1"[0])insert(<Індекс>, <Число>)
```

Видалення елементів

```
c.pop()  pop([<Індекс>])
del c[:2], del c[5] по елементу або зрізу
c.remove(b"s"[0]) remove(<Число>)
```

Інші методи

`c.reverse()` `reverse ()`

c.decode("cp1251") b.decode(encoding="cp1251", errors="replace")

Перетворення об'єкта в послідовність байтів. Модуль **`pickle`**

Модуль pickle реалізує потужний алгоритм серіалізації і десеріалізації об'єктів Python.

"Pickling"(серіалізація) - процес перетворення об'єкта Python в потік байтів.

"`Unpickling`" (десеріалізація)- зворотна операція, в результаті якої потік байтів перетворюється назад в Python-об'єкт.

Так як потік байтів легко можна записати в файл, модуль pickle широко застосовується для збереження і завантаження складних об'єктів в Python.

3. Перш ніж використовувати функції із цього модуля, необхідно підключити модуль за допомогою інструкції:

```
import pickle
```

Для перетворення призначено дві функції:

`dumps` і `loads`.

Функція `dumps`

1. Виконує *сериалізацію* об'єкта й повертає послідовність байтів спеціального формату.

`dumps``(<Об'єкт>[,<Протокол>])`

2. Формат залежить від зазначеного протоколу: число від 0 до 4 (підтримка протоколу 4 з'явилася в Python 3.4).

3. Якщо другий параметр не зазначений, буде використаний протокол 4 для Python 3.4 або 3 – для попередніх версій Python 3.

Приклад 4.203. Приклад перетворення списку й кортежу:

```
>>> import pickle
>>> obj1 = [1, 2, 3, 4, 5]  # Список
>>> obj2 = (6, 7, 8, 9, 10) # Кортеж
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
```

```
>>> pickle. dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

Функція `loads`

```
pickle.loads(pickle object)
```

1. Перетворює послідовність байтів спеціального формату назад в об'єкт, виконуючи його *десеріалізацію*.

Приклад відновлення списку й кортежу:

Приклад 4.204.

```
>>>pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

Шифрування рядків

Для шифрування рядків призначений модуль `hashlib`.

1. Перш ніж використовувати функції із цього модуля, необхідно підключити модуль за допомогою інструкції:

```
import hashlib
```

2. Модуль надає наступні функції:

`md5(), sha1(), sha224(), sha256(), sha384() і sha512().`

3. Як необов'язковий параметр функціям можна передати послідовність байтів, яка має бути зашифрована.

Приклад 4.205.

```
>>> import hashlib
>>> h = hashlib.sha1(b"password")
```

`update().` Передати послідовність байтів можна також за допомогою методу `update()`. У цьому випадку об'єкт приєднується до попереднього значення:

```
>>> h = hashlib.sha1()
>>> h.update(b"password")
```

Методи: **`digest()`** *і* **`hexdigest( )`**

Одержати зашифровану послідовність байтів і рядок дозволяють два методи: `digest()i hexdigest()`.

1. Перший метод повертає значення типу `bytes`,
2. Другий метод повертає рядок, що містить шістнадцяткові цифри.

Приклад 4.206.

```
>>> h = hashlib.sha1(b"password")
>>> h.digest()
b'[\xaaa\xe4\xc9\xb9??\x06\x82%\x0bl\xf83\x1b~\xe6\x8f\
xd8'
>>> h.hexdigest()
'5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'
```

Функція **`md5()`**

Найбільш часто застосовуваною є функція `md5()`, яка шифрує рядок за допомогою алгоритму MD5.

Ця функція використовується для шифрування паролів.

Для порівняння введеного користувачем пароля зі збереженим у базі необхідно:

зашифрувати введений пароль,

потім виконати порівняння.

Приклад 4.207. Перевірка правильності введення пароля

```
>>> import hashlib
>>> h = hashlib.md5(b"password")
>>> p = h.hexdigest()
>>> p # Пароль, збережений у базі
'5f4dcc3b5aa765d6ld8327deb882cf99'
>>>h2 = hashlib.md5(b"password")
# Пароль, введений користувачем
>>> if p == h2.hexdigest():print("Пароль правильний")
```

Атрибут `digest_size` зберігає розмір значення, згенерованого описаними раніше функціями шифрування, у байтах:

```
>>> h = hashlib.sha512(b"password")
>>> h.digest_size
64
```

Починаючи з Python 3.4 підтримує новий спосіб стійкого до зламу шифрування паролів за допомогою функції `pbkdf2_hmac()`:

```
pbkdf2_hmac(<Основний алгоритм шифрування>, <Пароль, що зашифровують>, <"Сіль">, <Кількість проходів шифрування>, dklen=None)
```

Як `<основний алгоритм шифрування>` слід указати рядок з найменуванням цього алгоритму: `"md5"`, `"sha1"`, `"sha224"`, `"sha256"`, `"sha384"` і `"sha512"`.

`<Пароль, що зашифровують>` вказується у вигляді значення типу `bytes`.

`"Сіль"` – це особлива величина типу bytes, що виступає як ключ шифрування, – її довжина не повинна бути меншою за 16 символів.

Кількість проходів шифрування слід указати достатньо великою (так, при використанні алгоритму SHA512 вона повинна становити 100000).

ПРИМІТКА

Кодування даних із застосуванням функції `pbkdf2_hmac()` вимагає багато системних ресурсів і може забрати значний час, особливо на малопотужних комп'ютерах.

Параметр `dklen` функції `pbkdf2_hmac()` вказує довжину результуючого закодованого значення в байтах – якщо вона не задана або дорівнює None, буде створено значення стандартної для вибраного алгоритма довжини (64 байта для алгоритма SHA512).

Закодований пароль повертається у вигляді величини типу `bytes`.

Приклад 4.208.

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha512', b'1234567', b'saltsaltsaltsalt', 100000)
>>> dk
```

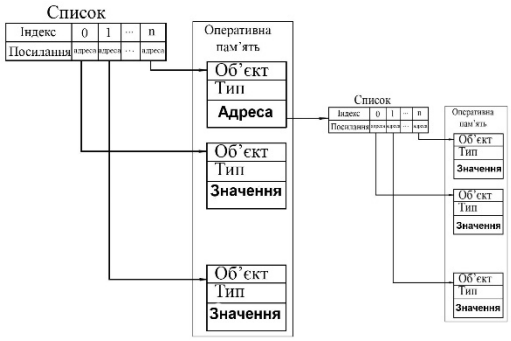
b"Sb\x85tc-\xcb@\xc5\x97\x19\x90\x94@\x9f\xde\x07\xa4p-\x83\x94\xf4\x94\x99\x07\xec\xfa\xf3\xcd\xc3\x88jv\xd1\xe5\x9a\x119\x15/\xa4\xc2\xd3N\xaba\x02\xc0s\xc1\xd1\x0b\x86xj(\x8c>Mr'@\xbb"

4.5. 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: $A = \left[\left[\left(1,2\right),3\right],\ \left[\left\{15,18\right\},\ \left[\left\{"d":19\right\},m\right]\right]\right]$

Розглянемо список $L = \left[4,5,...,\left[1,2,3\right]\right]$



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`).

Списки

1.Списки є змінюваними типами даних.

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

Приклад 4.209.

```
>>> arr1=arr = [1, 2, 3] # Створюємо список
>>> arr[0]       # Одержуємо елемент по індексу
1
>>> arr[0] = 50 # Змінюємо елемент по індексу
>>> arr
[50, 2, 3]
>>> arr1 is arr
True
```

Кортежі

1.Кортежі є незмінюваними типами даних.

Іншими словами, можна одержати елемент по індексу, але змінити його не можна:

Приклад 4.210.

```
>>> 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

1. 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).

– елементи множини не індексуються.

Приклад 4.211. Створення множини

Створення зі списку: **set()**

```
>>> 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*)**

4. 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 4.212.

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
```

4.5.1. Способи 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 4.213.

```
>>> 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 4.214.

```
>>> 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 4.215.

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

Відмінність від PHP

У деяких мовах програмування (наприклад, у PHP) можна додати елемент, указавши порожні квадратні дужки або індекс більший, ніж останній індекс. У мові Python усі ці способи призведуть до помилки:

```
>>> arr = []  #Створюємо список
>>> arr[] = 10 #Помилкова операція
    arr[] = 10
        ^
Syntaxerror: invalid syntax
arr+=[10]      # правильно
arr.append(1)  # правильно
>>> arr = []#Створюємо список
>>> arr[0] = 10 #Помилкова операція
Indexerror: list assignment index out of range
arr=[10]  # правильно
```

Особливості створення списку

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 4.216.

```
>>> 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 4.217.

Тепер спробуємо змінити значення в змінній y:

```
>>> y[1] = 100  # Змінюємо другий елемент
>>> 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 позиційне присвоювання:

Приклад 4.218.

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

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 4.219.

```
>>>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]]
```

Створення вкладених списків

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

Приклад 4.220.

```
>>> 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]]
```

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 4.221.

```
>>> arr0 = [ i**2 for i in range(3) ]
>>> arr0
[0, 1, 4]
>>> arr = [ [i**2] for i in range(3) ]
>>> 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]
```

Перевірка посилань змінних

1. Перевірити, чи посилаються дві змінні на той самий об'єкт, дозволяє оператор `is`.

2. Якщо змінні посилаються на той самий об'єкт, то оператор `is` повертає значення `True`:

Приклад 4.222.

```
>>> x = y = [[1,[3]], [2,[4]]] # Неправильно
```

```
>>> x is y
True
>>>x,y = [[1,[3]], [2,[4]]],[[1,[3]], [2,[4]]] >>> x is y   # Це різні об'єкти
False
```

4.5.2. Спопсоби копіювання списків

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 4.223. 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()`:**

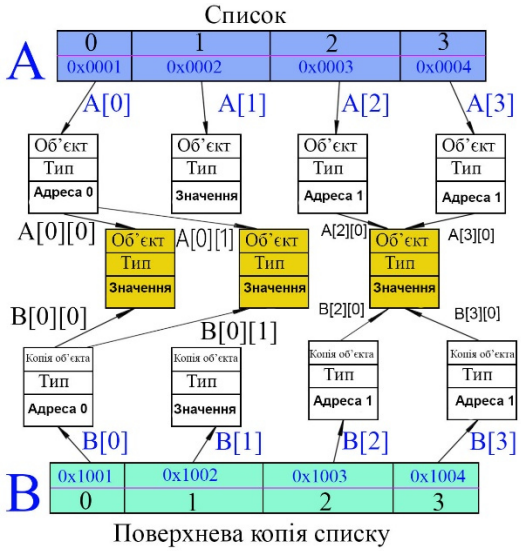
```
>>> x = ["2017","2018","2019","2020","2021",]
>>> y= x.copy()
>>> y
['2017', '2018', '2019', '2020', '2021']
>>> x is y  # Оператор показує, що це різні об'єкти
False
>>> y[4] = "2022" # Змінюємо елемент
>>> x
['2017', '2018', '2019', '2020' ,'2021']
>>>y # Змінився тільки список у змінній y
['2017', '2018', '2019', '2020','2022']
```

Поверхнева копія списку

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

2. Розглянуті методи створення копії списку створюють поверхневу копію.

```
x=y=[4]

A=[[1,2],3, x, y]
B=list(A)
print("Список A:", A)
print("Копія  B:", B)
print("A[0] is B[0]:", A[0] is B[0])

Результат:

Список A: [[1, 2], 3, [4], [4]]

Копія  B:  [[1, 2], 3, [4], [4]]

A[0] is B[0]:  True
```

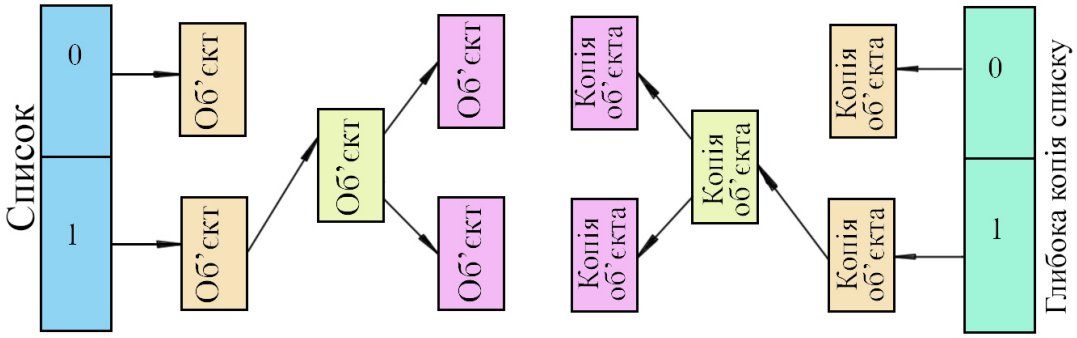


Означення поверхневої та глибокої копії

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 4.224.

```
>>> 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 4.225.

```
>>> 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 4.226.

```
>>>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 y[0], z[1] is y[1], 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]
```

4.5.3. 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 4.227. Вивід елементів списку:

```
>>>r=range(5,20,2)
>>>arr = [r, "str", 4.1, "5"]
>>>arr[0][3]
11
>>>arr.append(10)  # Додає елемент і нічого не повертає
>>>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 = повинні збігатися, інакше буде виведене повідомлення про помилку:

Приклад 4.228.

```
>>> 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 4.229.

```
>>>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 4.230.

```
>>> 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 4.231.

```
>>> 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]
```

Від'ємний індекс

1. Як індекс можна вказати від'ємне значення.

2. Зсув буде відлічуватися від кінця списку.

Приклад 4.232.

```
>>> 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 4.233.

```
>>> 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 4.234. 0 1 2 3 4 5

```
>>> 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
```

Приклад 4.235.

Виведемо список у зворотному порядку

```
>>> 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 4.236.

```
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 4.237.

```
>>> 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 4.238.

```
>>> 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 4.239.

```
>>>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)
```

Багатовимірні списки

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

Наприклад, елемент списку може бути:

- числом,

- рядком,

- списком,

- кортежом,

-словником і т. д.

Створити вкладений список можна, наприклад, так:

Приклад 4.240.

```
>>>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 4.241.

```
>>> 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 4.242.

```
>>> 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 4.243.

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'
```

4.5.4. Перебір елементів списку

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

Змінна `i` містить поверхневу копію елемента списку.

Тому модифікація змінної в тілі циклу не впливає на елементи списку.

Приклад 4.244.

```
arr = [1, 2, 3, 4, 5, 6]
for i in arr:
    i+=10
    print(i, end=" ")
print("\n",arr)
```

Результат:

```
11 12 13 14 15 16
 [1, 2, 3, 4, 5, 6]
```

При модифікації рядкової змінної елементи списку також залишаються незмінними

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

Результат:

```
U - k - r - a - i - n - e -
 ['U', 'k', 'r', 'a', 'i', 'n', 'e']
```

В даному випадку змінна циклу посилається на незмінний тип даних

Модифікація змінної циклу може призводити до зміни початкового списку

```
arr = [[1, 2], [3, 4]]
print("Початковий список", arr)
for i in arr: i[0] += 10 #
print("Змінений список",arr)
```

Результат:

```
Початковий список [[1, 2], [3, 4]]
Змінений список [[11, 2], [13, 4]]
a=bytearray("My info","cp1251")
b=bytearray("My string","cp1251")
arr=[a,b]
print(arr)
for i in arr:
    i+=bytearray(" changed","cp1251")
print(arr)
```

Результат:

[[bytearray(b'My info'), bytearray(b'My string')]

[bytearray(b'My info changed'), bytearray(b'My string changed')]

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.

Приклад застосування функції **range()**

Приклад 4.245.

Помножимо кожний елемент списку на 2:

```
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]
```

i	Дія	arr
0	-	1
1	2+1	3
2	3+3	6
3	4+6	10
4	5+10	15
5	6+15	21
6	7+21	28

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 4.246.

```
>>> 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')]
```

Приклад застосування функції **enumerate().**

Помножимо кожний елемент списку на 2:

Приклад 4.247.

```
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`.

Використовуючи цикл `while` знайдемо такий `arr1,` для якого `arr[3]-arr1[3]==3`

Приклад 4.248.

```
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]

4.5.5. Генератори списків

У **прикладі 3** 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 4.249. 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 4.250.

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

Приклад 4.251. 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]
```

Ускладнимо приклад

Одержимо парні елементи вкладеного списку й помножимо їх на 10:

Приклад 4.252.

```
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 4.253. 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]
```

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 4.254.

```
>>> 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 4.255.

```
>>> 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

4.5.6. Функції-ітератори для списків

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

2. Параметр < посилання на функцію> містить посилання на функцію, у яку будуть передаватися поточні елементи послідовності, що розміщені на одному зсуві.

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

Приклад використання функції **map()**

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

Приклад 4.255. 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 4.256. 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 4.257. 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 4.258. 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 4.259. 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]

Знайдемо суму елементів трьох списків

Приклад 4.260.

```
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 4.261.

```
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]
```

map *і кілька послідовностей різної довжини*

Якщо кількість елементів у послідовностях буде різною, то в якості обмеження вибирається послідовність із мінімальною кількістю елементів:

Приклад 4.262.

```
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]
```

Вбудована функція `zip()` *(застібка-блискавка)*

Формат функції:

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

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

Приклад 4.263.

```
>>> 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 4.264.

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

Приклад 4.265.

```
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 4.266.

```
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 4.267.

```
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))#!!!!!!!!
[(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 4.268.

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

Приклад 4.269.

```
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 4.270.

```
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 4.271.

```
arr1 = [1, 2, 3, 4, 5]       # 5 елементів
arr2 = [10, 20, 30, 40, 50] # 5 елементів
arr3 = [100, 200, 300, 400] # 4 елементи
arr=[min(d) for d in zip(arr1,arr2,arr3)]
print(arr)# Результат:[1, 2, 3, 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 4.272. 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 4.273.

```
>>> [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 4.274. 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 4.275. 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]
```

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 4.276. 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('вариант1',mysuma)
# Результат виконання:
(5, 1) (6, 2) (8, 3) (11, 4) (15, 5) вариант1 20
mysuma = functools.reduce(fred, arr)
print('вариант2',mysuma)
```

```
# Результат виконання:
(1, 2) (3, 3) (6, 4) (10, 5) вариант2 15
mysuma = functools.reduce(fred, [], 10)
print('вариант3',mysuma)
# Результат виконання: вариант3 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('вариант1',mysuma)
# Результат виконання:
(1, 2) (2, 3) (6, 4) (24, 5) вариант1 120
mysuma = functools.reduce(fred, arr, 10)
print('вариант2',mysuma)
# Результат виконання:
(10, 1) (10, 2) (20, 3) (60, 4) (240, 5) вариант2 1200
mysuma = functools.reduce(fred, [1,2], 3)
print('вариант3',mysuma)
# Результат виконання:
(3, 1) (3, 2) вариант3 6
```

Приклад застосування ітераторів

```
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
```

Результат:

40

[9, 0, 36, 0, 4]

[36, 4]

```
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])))
```

4.5.7. Методи модифікації списків

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 4.277.

```
>>>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 4.278.

```
>>> 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 4.279.

```
>>> 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 4.280. 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 4.281.

```
>>> 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 4.282.

```
>>> 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 4.283.

```
>>> 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 4.284.

```
>>>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 4.285.

```
>>> 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.

Приклад 4.286.

```
>>> 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 4.287.

```
>>> 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]
```

4.5.8. Функції для аналізу списків

Функція **any()**

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 4.288.

```
>>> any([0, None])
False
>>> any([])
False
>>> any([0, None, 1])
True
>>> any(["a"])
True
```

Функція **all**

1. Функція **all (<Послідовність>)** повертає значення **True**, якщо всі елементи послідовності в логічному контексті повертають значення **True** або послідовність не містить елементів взагалі.

Приклад 4.289.

```
>>> all([18, "Petrenko"])
True
>>> all([])
True
>>> all([0, "Petrenko"])
False
>>> all([18, ""])
False
>>> all([0, None])
False
```

Перевертання списку. Метод **reverse()**

Метод `reverse()` змінює порядок проходження елементів списку на протилежний. Метод змінює поточний список і нічого не повертає.

Приклад 4.290.

```
>>> 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 4.291. 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]
```

Сортування списку

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 4.292.

```
>>> 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 4.293.

```
>>> 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 4.294.

```
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 4.295.

```
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 4.296. Сортування по першому елементу

```
>>> def getkey(item):
        return item[0]+item[1]
>>> s = [[10, 3], [1, 7], [9, 34], [3, 64]]
>>> s.sort(key=getkey)
>>> s
[[1, 7], [10, 3], [9, 34], [3, 64]]
```

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 4.297. 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 4.298.

```
>>> 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 4.299.

```
>>> arr = [ "word1", "word2", "word3", 2]
>>> str(arr)
"['word1', 'word2', 'word3', 2]"
```

Для створення кортежу необхідна кома

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 4.300.

```
>>> 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.

4.5.9. Створення множин та операції над множинами

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`**`()`:

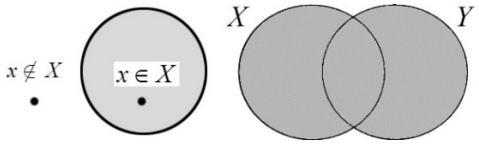
Приклад 4.301.

`>>>s=`**`set`**`(3,4)`
`>>> s`
`{3, 4}`
	`>>>s=`**`set`**`([1,2])`
	`>>>s`
	`{1, 2}`
		`>>>s=`**`set`**`("ab")`
		`>>>s`
		`{'b', 'a'}`

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

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

Оператори й методи для роботи з множинами



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

Приклад 4.302.

```
>>> 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

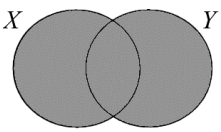
Якщо даний елемент уже міститься в іншій множині, то він повторно доданий не буде:

Приклад 4.303.

```
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}
ножина 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 4.304.

```
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}
```

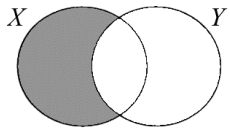
Приклад 4.305.

```
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 4.306.

```
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:

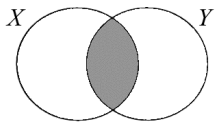
Приклад 4.307.

```
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 4.308.

```
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)**

Перетин

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

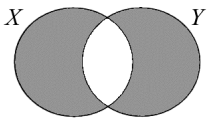
Приклад 4.309.

```
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 4.310.

```
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 4.311.

```
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 4.312.

```
>>> 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`.

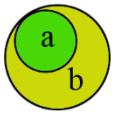
Приклад 4.313.

```
>>> 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 4.314.

```
>>> 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:

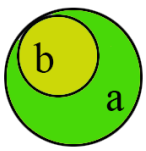
Приклад 4.315.

```
>>> 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 4.316.

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

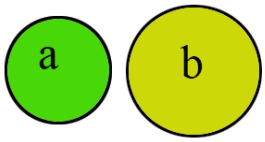


```
False
>>> s = set (["a", "b", "c"])
>>> s > set(["a", "b", "c"])
False
>>> s > set(["a", "b"])
True
```

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

Приклад 4.317.

```
>>> 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
```



4.5.10. Методи копіювання та модифікації множин

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

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

Приклад 4.318.

```
>>> 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 4.319.

```
>>> s = set([1, 2, 3])
>>> s.add(4); s
{1, 2, 3, 4}
```

```
my= {"Petrenko"}
my^= {"Ivan"}
my^= {"Ivanovich"}
print (my)
```

```
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 4.320.

```
>>> 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 4.321.

```
>>> 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 4.322.

```
>>> 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
```

4.5.11. 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 4.323.

```
>>> {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 4.324. 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 4.325

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

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

Приклад 4.326.

```
>>> 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})
```

До множин типу `frozenset` застосовні ті оператори, які їх не змінюють. До них застосовні також наступні методи:

```
copy(),                    #Копіювання
difference(),              #Різниця
intersection(),            #Перетин
issubset(),                #Чи є підмножиною
issuperset(),              #Чи включає множину
symmetric_difference()     #Симетрична різниця
union().                   #Об'єднання
```

4.5.12. 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 4.327.

```
>>> 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})
```

Діапазони підтримують:

- доступ до елемента по індексу,

Index	0	1	2	3	4	5	6	7	8
Value	1	2	3	4	5	6	7	8	9

```
>>> r = range(1, 10)
>>> r[2], r[-1]
(3, 9)
```

- одержання зрізу (у результаті повертається також діапазон),

```
r = range(10)
nr= r[2:4]
print(type(nr))
a=list(nr)
print(a)
```

Результат

```
<class 'range'>
[2, 3]
```

- перевірку на входження

```
>>> 5 in range(10)
>>> True
```

Застосування функцій і методів з типом range

Приклад 4.328.

- застосування функції `len()`

```
>>>len(range(20,3,-2)) #20,18,16,14,12,10,8,6,4
9
```

- застосування функції `min()`

```
>>> min(range(20,3,-2)) # 20,18,16,14,12,10,8,6,4
4
```

- застосування функції `max()`

```
>>> max(range(0,127,3)) # 126
126
```

- метод `index()`

```
>>> range(0,127,3).index(15)
5
```

- метод `count()`

```
>>> range(0,127,3).count(16)
0
```

4.5.13. Оператори порівняння діапазонів

Оператор == – повертає `True`, якщо діапазони рівні, і `False` якщо ні. Діапазони вважаються рівними, якщо вони містять однакові послідовності чисел!!

Приклад 4.329.

```
>>> 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 4.330.

```
>>> 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 4.331.

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

4.5.14. 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 4.332

```
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 4.333.

```
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 4.334
```
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')]
```

4.6. Словники

4.6.1. Способи створення 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. Словник має такі ознаки:

Значення можуть мати довільний тип даних

Словник може 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.

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 4.335. 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 4.336

```
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 4.337.

```
>>> 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 4.338.

```
>>> 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 4.339

```
>>> 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 4.440.

```
# 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}
```

4.6.2. Способи копіювання словників

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 4.441.

```
# 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 4.442.

```
>>> 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()**

Фмат: <Словник2> = **dict**(<Словник1>)

Приклад 4.443.

```
>>> d1 = {"a": 1, "b": 2} # Створюємо словник
>>> d2 = dict(d1) # Створюємо поверхневу копию
>>> d1 is d2 # це різні об'єкти
False
>>> d2["b"] = 10
>>> dl, d2 # Змінилася тільки змінна d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Створити поверхневу копію словника методом **copy()**

Приклад 4.444.

```
>>> 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 4.445

```
>>> 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 4.446.

```
>>> import copy
>>> d3 = copy.deepcopy(d1) #створюємо глиб. копію
>>> d3["b"][1] = 800
>>> d1
({'a': 1, 'b': ['test', 30, 40]},
>>> d3 # Змінилося значення тільки в змінній
{'a': 1, 'b': ['test', 800, 40]})
```

4.6.3. Методи доступу 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 4.447.

```
>>> 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 4.448.

```
>>> 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 4.449.

```
>>> 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 4.450.

```
>>> 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 4.451.

```
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 4.452.

```
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}
```

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 4.453.

```
>>> 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'}
```

4.6.4. Визначення довжини словника, вилучення елемента словника

Визначення довжини словника

Одержати кількість ключів у словнику дозволяє функція `len()`:

Приклад 4.454.

```
>>> 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 4.455.

```
d = { "a": 1, "b": 2}
del d["b"]
print(d)  # Видаляємо елемент з ключем "b"
{'a': 1}
```

4.6.5. Перебір елементів та сортування по ключах

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 4.456.

```
#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 4.457.

```
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 4.458.

```
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)`

Тому що 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)`

4.6.6.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 4.459.

```
# 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 4.460.

```
>>> 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 4.461.

```
>>> 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 4.462.

```
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 4.463.

```
>>> 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 4.464.

```
>>> 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 4.465.

```
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}
```

4.6.7. 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 4.466.

```
# 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 зі 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 4.467.

```
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']
```

Контрольні запитання до розділу 4.

1.Які функції використовують для створення об’єктів цифрового типу?

2. Які математичні функції ви знаєте?

3. Опишіть призначення і поширені функції модуля math.

4. Створіть код алгоритму генератора паролів довільної довжини.

5. Які способи створення рядків ви знаєте?

6. Опишіть методи створення об’єктів типу **`bytes`** та **`bytearray`.**

7. Як створити рядок документування? Призначення рядка документування.

8. Які методи модифікації об’єктів рядкового типу вам відомі?

9. Опишіть три способи форматування рядків.

10. Яку спільну характеристику мають функції `map()` та `zip()`.

5. ФУНКЦІЇ КОРИСТУВАЧА

5.1. Визначення функцій

Функція – це фрагмент коду, який можна викликати з будь-якого місця програми **більше одного разу**.

У попередніх лекціях ми вже багато раз 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.

5.1.1. Структура і виклик функції

Функцію створюють (або, як говорять програмісти, визначають) за допомогою ключового слова

`def` за наступною схемою:

Приклад 5.1.

```
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 5.2.

```
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 5.3.

```
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 взагалі.

У цьому випадку виконуються всі інструкції усередині функції, і як результат повертається значення **None**.

5.1.2. Приклади визначення функцій.

Приклад 5.4.

```
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.

Приклади виклику функцій

Приклад 5.5.

```
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
```

5.1.3. Параметри у виклику та у визначенні

Ім'я змінної у виклику функції **може не збігатися** з іменем змінної у визначенні функції.

Приклад 5.6.

```
def echo(m) :
    print(m)
```

```
n="Параметр має інше ім'я"
echo(n)
```

Результати роботи:

```
Параметр має інше ім'я
```

5.1.4. Глобальні та локальні змінні

Крім того, *глобальні* змінні **x** и **y** не конфліктують з однойменними змінними у визначенні функції, оскільки вони розташовані в різних областях видимості.

Приклад 5.7.

```
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 5.8.

```
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
```

5.1.5. 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 5.9. 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 5.10. 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

5.1.6. Атрибути об'єкта типу **function**

Звернутися до атрибутів можна, указавши атрибут після назви функції через крапку.

риклад. 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 5.11. Атрибути функції **suma**

```
>>> def suma(x, y):
    """ Додавання двох чисел """
    return x + y
# 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__
```

5.1.7. Розташування визначення функцій

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 5.12.

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 самому початку програми після підключення модулів або в окремому модулі

5.1.8. Кілька функцій з однією назвою

За допомогою оператора розгалуження **if** можна вибирати необхідне визначення функції з однаковою назвою, але різною реалізацією.

Приклад 5.13.

```
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
```

5.1.9. Випадок перевизначення функції

Пам’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 раз, то буде використовуватися функція, яка була визначена останньою.

Приклад 5.14.

```
def echo():
    print("Ви ввели число 1")
def echo():
    print("Альтернативна функція")
echo() # Завжди виводить "Альтернативна функція"
```

```
Результат роботи: Альтернативна функція
```

5.2. Параметри функцій

5.2.1. 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 5.15. 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.

5.2.2. Позиційна передача параметрів

Позиційна передача параметрів – це присвоєння значень параметрам функції в порядку, у якому вони задані при виклику функції.

Приклад 5.16. 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 .

5.2.3. Передача параметрів зіставленням по ключах

1. При виклику функції параметрам можуть присвоюватися значення.
2. Послідовність вказівки параметрів у цьому випадку може бути довільною.

Приклад 5.17. 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 5.18.

```
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
```

5.2.4. 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 5.19.

```
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
```

5.2.5. 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)

5.2.6. Розпакування словника

Якщо значення параметрів містяться в словнику, то розпакувати словник можна, указавши перед ним дві зірочки: **(* *).**

Приклад 5.20.

```
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
```

5.2.7. 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 5.21.

```
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:

5.2.8. Передача у функцію змінюваних об'єктів

Приклад 5.22.

```
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.

Як уникнути зміни об'єкта?

Якщо необхідно уникнути зміни значень, усередині функції слід створити копію об'єкта.

Приклад 5.23. 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 5.24.

```
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}
```

5.2.9. 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 5.25.

```
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 5.26.

```
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]
```

Складність функцій

Функція може бути будь-якої складності і повертати будь-які об'єкти (списки, кортежі, і навіть функції!):

Приклад 5.27.

```
def outfunc(n):
    def infunc(x):
       return x + n
```

```
    return infunc
new = outfunc(100)  # new - это функция
print(new(200))
```

Результат:

300

Приклад 5.28. 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 5.29.

```
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

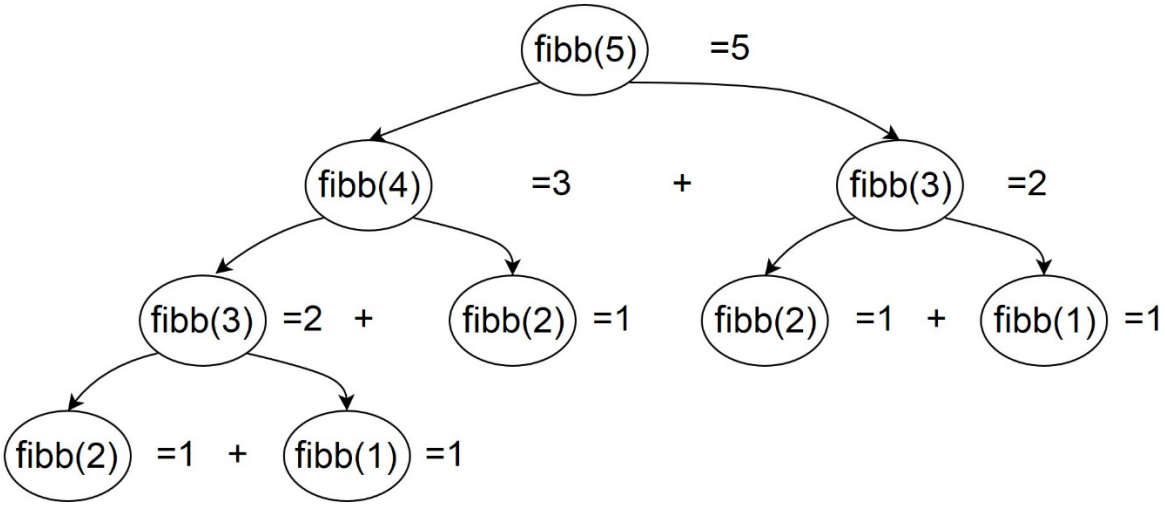
```
print(fibb(3))
```

Результат:2

```
print(fibb(4))
```

Результат:3

Граф виконання рекурсії



Приклад 5.30. 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:

```
30414093201713378043612608166064768844377641568960512000000
000000
```

5.2.10. Число параметрів функції. Фіксоване число параметрів у функції

Число параметрів, які використовуються у функції дорівнює числу параметрів, які вказані у круглих дужках.

Приклад 5.31.

```
def fixparam(a,b,c):
    return a+b-c
print(fixparam(1,2,3))
```

Результат: 0

Приклад 5.32.

```
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 5.33. 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 5.34. Збереження кортежів

```
def f(*t): print(t)
d=(1,2,3)
f(*d)
```

Результат роботи: (1, 2, 3)

5.2.11. Поєднання обов'язкових параметрів з зірковим

Спочатку вказують обов'язкові (позиційні) параметри, а останнім вказують параметр з зірочкою:

Приклад 5.35.

```
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 5.36.

```
def sumator(x, y=5, *t): # Комбінація параметрів
    res = x + y
    for i in t:
        res += i
    return res
print(sumator(10),end=" ")
print(sumator(10,20),end=" ")
# Перебираємо кортеж з переданими параметрами
print(sumator(10, 20, 30, 40, 50))
```

Результат: 15 30 150

5.2.12. Збереження іменованих параметрів у словнику

Якщо перед параметром у визначенні функції вказати дві зірочки: (**), то всі іменовані параметри будуть збережені в словнику.

Приклад 5.37. 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}
```

5.2.13. Поєднання обов'язкових параметрів з двозірковим

Спочатку вказують обов'язкові параметри, а останнім вказують двозірковий параметр:

Приклад 5.38.

```
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}
d=dicmix( a=5,k=1,b=2,c=3)
Результат: 5 {'k': 1, 'b': 2, 'c': 3}
```

```
def dicmix(a,b=2,**d):
    print(a, d, b)
d=dicmix( a=10,b=3,k=1,c=3)
Результат:
10 {'k': 1, 'c': 3} 3
```

Приклад 5.38.

```
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 5.39. 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, d )
func(10)
Результат роботи:
(10,) {}
func(a= 1, b=2)
Результат роботи:
() {'a': 1, 'b': 2}
func(20 , a= 1, b=2)
Результат роботи:
(20,) {'a': 1, 'b': 2}
```

Поєднання обов'язкових параметрів з зірковими

Спочатку вказують обов'язкові параметри, потім однозірковий параметр, а останнім вказують двозірковий параметр.

Приклад 5.40.

```
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}

5.2.14. 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 5.41. `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 визначені 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, щоб один з параметрів передавався обов'язково по імені

У цьому випадку вказують тільки зірочку без змінної:

Приклад 5.42.

```
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:

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)

5.3. Анонімні функції, функції-генератори, декоратори

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.

ЦЕ ОСНОВНА ВІДМІННІСТЬ LAMBDA ФУНКЦІЇ !!

```
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 5.43. 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
```

5.3.1. 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 5.44. 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
```

5.3.2. 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 5.45. 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 5.46. 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])
```

5.3.3. 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 5.46. 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
```

5.3.4. 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 5.47. 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
```

5.3.5. 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»

5.3.6. 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 5.48. 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 5.49. 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
```

5.3.7. Декоратори функцій

Декоратори дозволяють змінити поведінку звичайних функцій – наприклад, виконати які-небудь дії перед та після виконання функції.

Приклад 5.50.

```
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   # Виклик декоратора
def unchangeable_function():    #Визначення функції
    print(«Це базова функція»)
unchangeable_function()# Виклик функції користувача
```

Приклад 5.51. 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("Виклик функції func()")
    return "x = {0}".format(x)
print(func(10))
```

```
Результат роботи:
Тут можна писати код декоратора
Виклик функції 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 5.52.

```
def deco(f):
    print(«Тут можна писати код декоратора»)
    return f
def func(x):
```

```
    print(«А тепер запустили функцію func()»)
    return print («x = {0}».format(x))
# викликаємо функцію func() через функцію deco()
deco(func)(10)
```

Результат роботи:

```
Тут можна писати код декоратора
А тепер запустили функцію func()
x = 10
```

5.3.8. Кілька декораторів

Перед визначенням функції можна вказати відразу кілька функцій-декораторів. Для прикладу обернемо функцію `func()` у два декоратори: `deco1()` і `deco2()`.

Приклад 5.53.

```
def deco1(f):
    print("Викликана функція deco1()")
    return f
def deco2(f):
    print("Викликана функція deco2()")
    return f


@deco1
@deco2
def func(x):
   return print("x = {0}".format(x))
func(10)
```

Результат роботи:

```
Викликана функція deco2()
Викликана функція deco1()
x = 10
```

Використання двох декораторів еквівалентно наступному коду:

```
deco1(deco2(func))(10)
```

Тут спочатку буде викликана функція `deco2()`, а потім функція `deco1()`. Результат виконання буде присвоєний ідентифікатору `func`.

Приклад використання декоратора разом з функцією-обгорткою

Суть програми: дозволити виконання функції тільки при правильно введеному паролі.

Приклад 5.54. Обмеження доступу за допомогою декоратора

Крок 1. Генеруємо закодований пароль

```
import hashlib
ps = input("Введіть пароль для кодування: ")
h1 = hashlib.md5(bytes(ps, "cp1251"))
p_saved= h1.hexdigest()
print("Закодований пароль:\n p=", repr(p_saved))
```

Результат роботи:

```
Введіть пароль для кодування: universe
Закодований пароль:
p_saved='7d9a0d11cb36e12a68817aff945390de'
```

Крок 2А. Створюємо декоратор

```
import hashlib
def test_passw(f):
    ps = input("Введіть пароль: ")
    def check():
        h2 = hashlib.md5(bytes(ps,"cp1251"))
        if p_saved == h2.hexdigest(): # Порівнюємо
            f()
        else:
           print("Доступ закритий")
    return check
@test_passw
def func():
    print("Доступ відкритий")
func()   # Викликаємо функцію
```

Результат роботи:

Введіть пароль: 1 Доступ закритий	Введіть пароль: universe Доступ відкритий

У цьому прикладі після символу @ зазначено функцію - декоратор.

@test_passw – це виклик декоратора

Декоратор підміняє посилання на функцію **func** посиланням на функцію **deco**

Якщо введений пароль є правильним, то викликається функція **f()**, яка визначена наступним кодом**.**

```
    def func():
        return "Доступ відкритий"
```

а якщо ні, то друкуємо повідомлення про неправильний пароль.

Крок 2Б. Створюємо декоратор у функції

```
import hashlib
ps = input("Введіть пароль: ")
def test_passw(p):
    def deco (f):
        h2 = hashlib.md5(bytes(p,"cp1251"))
        if p_saved == h2.hexdigest(): # Порівнюємо паролі
            return f
        else:
           return lambda: "Доступ закритий"
    return  deco#Повертаємо функцію-декоратор
@test_passw(ps)
def func():
    return "Доступ відкритий"
print(func())   # Викликаємо функцію
```

Результат роботи:

Введіть пароль: 1 Доступ закритий	Введіть пароль: universe Доступ відкритий

Рекурсія. Обчислення факторіала

Рекурсія – це можливість функції викликати саму себе. Рекурсію зручно використовувати для перебору об'єкта, що має заздалегідь невідому структуру, або для виконання невизначеної кількості операцій. Як приклад розглянемо обчислення факторіала

Приклад 5.55.

```
def factor(n):
    if not n:
        return 1
    else:
        return n * factor(n - 1)
#-----------------------------------
while True:
    z = input("Введіть число: ")
    if z.isdigit():
        b = int(z)
        break
    else:
        print("Ви ввели не число!")
print("%s!=%s"%(b,factor(b)))
```

Результат роботи: 6!=720

Процес рекурсивних обчислень 3!

```
factor(3);
3 * factor(2);
3 * 2 * factor(1);
3 * 2 * 1 * factor(0);
3 * 2 * 1 * 1;
3 * 2 * 1
3 * 2;
6;
```

1.Викликаємо `factor(3)`.

2. Число 3 повертає True у логічному контексті, тому переходимо до блоку “else”.

3.Хочемо перемножити числа 3*x повернути відповідь, але для цього потрібно знати друге число x.

4. Цей другий співмножник повертає x функція factorl (3-1) або factor(2).

5. Формуємо новий виклик factor з параметром 2.

6. Число 2 повертає True у логічному контексті, тому переходимо до блоку “else”.

7. Тому знову намагаємося перемножити 2*y .

8. Для того, щоб знайти y викликаємо factor(1).

9. Число 1 повертає True у логічному контексті, тому переходимо до блоку “else”.

10. Робимо ще одну спробу множення. На цей factor(0) зразу повертає 1.

12. Повертаємося у попередню функцію і результат виклику y =factor(1) множимо на 2 .

13. Повертаємося у попередню функцію і результат виклику x=factor(2) множимо на 3.

14.Функція factor(3) є функцією першого виклику, яка повертає результат.

Простіше за все для обчислення факторіала скористатися функцією factorial() з модуля math.

Ця функція не завжди доступна у всіх мовах програмування. Тому корисно знати простий рекурсивний алгоритм обчислення факторіалу.

Приклад 5.56.

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

>>> math.factorial(6)
720
```

5.4. Області видимості

5.4.1. Глобальні й локальні змінні

Глобальні змінні – це змінні, оголошені в програмі поза функцією. В Python глобальні змінні видимі в будь-якій частині модуля, включаючи функції.

Приклад 5.57. Глобальні змінні.

```
def func (glob2):
    print("Значення глобальної змінної glob1 =", glob1)
    glob2 += 10    #Це є локальна змінна
    print("Значення локальної змінної glob2 =", glob2)
glob1, glob2 = 10, 5
func(77) # Викликаємо функцію
print("Значення глобальної змінної glob2 =", glob2)
```

Результат виконання:

```
Значення глобальної змінної glob1 = 10
Значення локальної змінної glob2 = 87
Значення глобальної змінної glob2 = 5
```

Змінній `glob2` усередині функції присвоюється значення, передане при виклику функції.

У результаті створюється нова змінна з тим же ім’ям: glob2, яка є локальною.

Усі зміни цієї змінної усередині функції не торкнуться значення однойменної глобальної змінної.

Отже, *локальні змінні* – це змінні, що оголошуються усередині функцій.

Якщо ім'я локальної змінної збігається з іменем глобальної змінної, то всі операції усередині функції здійснюються з локальною змінною, а значення глобальної змінної не змінюється.

5.4.2. Видимість локальних змінних

Локальні змінні видимі тільки усередині тіла функції.

Приклад 5.58. Локальні змінні

```
def func():
    local1 = 77 # Локальна змінна
```

```
    glob1 = 25  # Локальна змінна
    print("Значення glob1 усередині функції =", glob1)
glob1 = 10   # Глобальна змінна
func()       # Викликаємо функцію
print("Значення glob1 поза функцією =", glob1)
try:
    print(local1)
except NameError: # Обробляємо виключення
    print("Змінна local1 не видима поза функцією")
```

Результат виконання:

Значення glob1 усередині функції = 25

Значення glob1 поза функцією = 10

Змінна local1 не видима поза функцією

Як видно з прикладу, змінна `local1`, оголошена усередині функції `func()`, недоступна поза функцією.

Оголошення усередині функції локальної змінної `glob1` не змінило значення однойменної глобальної змінної.

Якщо доступ до змінної в тілі функції здійснюється до присвоювання значення, то буде викликане виключення **`UnboundLocalError`**

Приклад 5.59. Помилка при доступі до змінної до присвоєння значення

```
def func():
    #Локальная змінна ще не визначена
    print(glob1)# Цей рядок викличе помилку!!!
    glob1 = 25 #За наявності цього рядка
glob1 = 10 # Глобальна змінна
func()     # Викликаємо функцію
Результат виконання:
UnboundLocalError: local variable 'glob1' referenced before
assignment
```

Для того, щоб значення глобальної змінної можна було змінити усередині функції, необхідно оголосити змінну глобальною за допомогою ключового слова **global**. Продемонструємо це на прикладі.

Приклад 5.60.

```
def func():
    # Оголошуємо змінну glob1 глобальною
    global glob1
    print("В тілі функції до зміни =", glob1)
    glob1 = 25 # Змінюємо значення глобальної змінної
    print("В тілі функції після зміни =", glob1)
glob1 = 10 # Глобальна змінна
print("glob1 до виклику функції =", glob1)
func() # Виклик функції
print("glob1 після виклику функції =", glob1)
Результат виконання:
    glob1 до виклику функції = 10
    В тілі функції до зміни = 10
    В тілі функції після зміни = 25
    glob1 після виклику функції = 25
```

Таким чином, пошук ідентифікатора, використовуваного усередині функції, буде проводитися в наступному порядку:

1. Пошук оголошення ідентифікатора усередині функції (у локальній області видимості).

2. Пошук оголошення ідентифікатора в глобальній області.

3. Пошук у вбудованій області видимості (вбудовані функції, класи і т. д.).

При використанні анонімних функцій слід враховувати, що при використанні усередині функції глобальної змінної буде збережено посилання на цю змінну, а не її значення в момент визначення функції:

Приклад 5.61.

```
x = 5
# Зберігається посилання, а не значення змінної x!!!
```

```
func = lambda: x
print(func())
Результат виконання:5
x = 80 # Змінили значення
print(func()) # Виведе: 80, а не 5
Результат виконання:80
```

Якщо необхідно зберегти саме поточне значення змінної, то можна скористатися способом, наведеним у прикладі.

Приклад 5.62. Збереження значення змінної

```
x = 5
# Зберігається значення змінної x
func = (lambda y: lambda: y)(x)
x = 80 # Змінили значення
print(func()) # Виведе: 5
Результат виконання: 5
```

Зверніть увагу на третій рядок прикладу. У ньому ми визначили анонімну функцію з одним параметром посилання, яка повертає посилання на вкладену анонімну функцію. Далі ми викликаємо першу функцію за допомогою круглих дужок і передаємо їй значення змінної x. У результаті зберігається поточне значення змінної, а не посилання на неї.

Зберегти поточне значення змінної можна також, указавши глобальну змінну як значення параметра за замовчуванням у визначенні функції.

Приклад 5.63. Збереження значення за допомогою параметра за замовчуванням

```
b = 5
 # Зберігається значення змінної x
func = lambda b = b: b
b = 80 # Змінили значення
print(func())
Результат виконання: 5
```

Одержати всі ідентифікатори і їх значення дозволяють наступні функції:

Функція `globals()` – повертає словник з глобальними ідентифікаторами;

Функція `locals()` – повертає словник з локальними ідентифікаторами.

Приклад 5.64.

```
def func():
   local1 = 54
   glob2 = 2
   print("Локальні ідентифікатори усередині функції")
   print(sorted(locals().keys()))
glob1, glob2 = 10, 88
func()
print ("Глобальні ідентифікатори поза функцією")
print(sorted(globals().keys ()))
```

Результат виконання:

Локальні ідентифікатори усередині функції

['glob2', 'local1']

Глобальні ідентифікатори поза функцією

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'func', 'glob1', 'glob2']

5.4.3. Вкладені функції

Одну функцію можна вкласти в іншу функцію, причому рівень вкладеності не обмежений. При цьому вкладена функція одержує свою власну локальну область видимості, має доступ до змінних, оголошених всередині функції, у яку вона вкладена (функції-предка). Розглянемо вкладення функцій на прикладі.

Приклад 5.65.

```
def func1 (b) :
    def func2():
        print(b)
    return func2
f1 = func1(10)
f2 = func1(99)
```

```
f1()
```
Результат виконання: 10
```
f2()
```
Результат виконання: 99

Розглянемо приклад

Тут ми визначили функцію `func1()`, що приймає один параметр `b`.

```
def func1 (b) :
    def func2():
        print(b)
    return func2
```

Усередині функції `func1()` визначена вкладена функція `func2()`.

```
def func2():
    print(b)
```

Результатом виконання функції `func1()` буде посилання на цю вкладену функцію.

```
return func2
```

Усередині функції `func2()` ми виконуємо вивід значення змінної `b`, яка є локальною у функції `func1()`.

```
print(b)
```

Розглядаємо області видимості

1. Глобальна область видимості
2. Локальна область видимості.

Порядок пошуку індентифікаторів:

1. Пошук всередині вкладеної функції
2. Пошук всередині функції-предка,
3. Пошук у функціях більш високого рівня
4. Пошук у глобальній і вбудованих областях видимості.

Значення у глобальній області видимості

```
a=10
def b():
    def d(): print(a)
```

```
    d()
b()
```

Результат: 10

У нашому прикладі змінна a буде знайдена в глобальній області видимості.

5.4.4. Значення у вбудованій області видимості

```
from math import *
def b():
    def d():
      def c():
        print("Pi =", round(pi, 2))
      c()
    d()
b()
```

Результат:

```
Pi = 3.14
```

Інструкція `from math import *` відкрила вбудовану область видимості модуля **`math`**.

Зберігаються лише посилання на змінні, а не їхні значення в момент визначення функції. У функції `func1()` надамо нове значення змінній `b`.
Тоді у функції `func2()` буде використовуватися це нове значення не залежно від місця його перевизначення:

Приклад 5.66.

```
def func1(b):
    def func2():
        print(b)
    b = 30
    return func2
f1 = func1(10)
f2 = func1(99)
```

```
def func1(b):
    b = 30
    def func2():
        print(b)
    return func2
f1 = func1(10)
f2 = func1(99)
```

```
f1 ()
```

```
Результат виконання: 30
f2 ()
Результат виконання: 30
```

Значення змінної буде збережено при визначенні вкладеної функції, якщо передати її значення як значення за замовчуванням до локальної зміни цього значення:

Приклад 5.67.

`def func1(b) :`
`  def func2(b=b):`
`  print(b)`
`  b = 30`
`  return func2`
`f1 = func1("рядок")`
`f2 = func1(100)`
`f1()`
`f2()`
	`def func1(b) :`
	`  b = 30`
	`  def func2(b=b):`
	`  print(b)`
	`  return func2`
	`f1 = func1("рядок")`
	`f2 = func1(100)`
	`f1()`
	`f2()`

рядок **100**	**30** **30**

5.4.5. Область видимості `nonlocal`

Область видимості `nonlocal` – це щось середнє між `global` і `local`. Цю область видимості використовуємо, якщо хочемо мати доступ до локальної змінної функції-предка.

Приклад 5.68.

```
def func_outer():
    x = 2
    print('x=', x)
    def func_inner():
        nonlocal x
        x = 5
    func_inner()
```

```
    print('Локальна змінна x змінена. Тепер x=', x)
func_outer()
```

Результат:

```
x= 2
Локальна змінна x змінена. Тепер x= 5
```

Зміна значення змінної `b`, оголошеної в функції **`func1()`** із вкладеної функції **`func2()`**.

Якщо в функції `func2()` присвоїти значення змінній `b`, то буде створена нова локальна змінна з таким же ім'ям.

Якщо усередині функції `func2()` оголосити змінну як глобальну й присвоїти їй значення, то зміниться значення глобальної змінної, а не значення змінної `b` усередині функції `func1()`.

Отже, жоден з вивчених раніше способів не дозволяє із вкладеної функції змінити значення змінної, оголошеної усередині функції-предка.

Щоб розв'язати цю задачу, слід оголосити необхідні змінні за допомогою ключового слова **`nonlocal`.**

Приклад 5.69.

```
def func1(a):
    print("a=",a, end=", ")
    b = a
    def func2(c):
        print("c=",c, end=", ")
        nonlocal b  # Оголошуємо змінну як nonlocal
        print("b=",b, end="|  ")
        b = c
    return func2
f = func1(10)
f(5)
f(12)
```

```
f(3)
f(120)
```

Результат виконання:

a= 10, c= 5, b= 10| c= 12, b= 5| c= 3, b= 12| c= 120, b= 3|

При використанні ключового слова **nonlocal** слід пам'ятати, що змінна обов'язково повинна існувати усередині функції-предка. В протилежному випадку буде виведене повідомлення про помилку.

Приклад 5.70.

Код з використанням nonlocal	**Код без nonlocal**

`def myfunc1():`
`  x = "Іван"`
`  def myfunc2():`
`  nonlocal x`
`  x = "Марія"`
`  myfunc2()`
`  return x`
`print(myfunc1())`
	`def myfunc1():`
	`  x = "Іван"`
	`  def myfunc2():`
	`  x = "Марія"`
	`  myfunc2()`
	`  return x`
	`print(myfunc1())`

Результат: Марія	**Результат: Іван**

5.5. Анотації функцій

В Python 3 функції можуть містити анотації, які вводять новий спосіб документування.

Тепер у заголовку функції допускається:

-вказувати призначення кожного параметра,

-дані якого типу він може приймати,

-тип значення, що повертається функцією.

Анотації мають наступний формат:

```
def <Ім'я функції>(
[<Параметр1>[: <Вираз>] [= <Значення за замовчуванням>]
[,...<Параметрn>[: <Вираз>][= <Значення за замовчуванням>]]]
```

) -> <тип значення, що повертається>:

<Тіло функції>

Анотації функцій, як для параметрів, так і для типу значень, що повертає функція, є абсолютно необов'язковими.

У параметрах

<Вираз> і < тип значення, що повертається >

<Вираз>- будь-який припустимий вираз мови Python.

Цей вираз буде виконано при створенні функції.

Приклад 5.71. Вказування анотацій.

```
def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
    print(a,"=",b)
func("Мій параметр")
```

Результат:

Мій параметр = 3

Для змінної a створений опис "Параметр1".

Для змінної b вираз 10 + 5 є описом.

Число 3 – значення параметра за замовчуванням.

Після закриваючої круглої дужки зазначений тип значення, що повертається функцією: None.

5.5.1. Атрибут **__annotations__**

Після створення функції всі вирази будуть виконані, і результати збережуться у вигляді словника в атрибуті __annotations__ об'єкта функції. Для прикладу виведемо значення цього атрибута:

Приклад 5.72.

```
def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
    print(a,"=",b)
func("Мій параметр")
print(func.__annotations__)
```

Результат:

Мій параметр = 3

```
{'b': 15, 'a': 'Параметр1', 'return': None}
```

Приклад 5.73.

```
def func(a, b, c):   # Неанотована функція
    return a + b + c
print(func(1, 2, 3))
```

Результат: 6

```
#Анотована функція
def func(a:'spam', b:(1,10),c:float) -> int:
  return a + b + c
print(func(1, 2, 3))
```

Результат: 6

Анотовані три аргументи.

Анотації для аргументів вказуються через двокрапку, відразу після імені аргументу.

Для значення, що повертається - після символів ->, слідом за списком аргументів.

Якщо в оголошенні функції присутні анотації, інтерпретатор збере їх в словник і приєднає його до об'єкту функції.

Імена аргументів стануть ключами,

Анотація, що повертається, буде збережена в ключі «return», значенням ключів цього словника будуть присвоєні результати виразів в анотаціях:

```
>>> func .__ annotations__
{'a':'spam','c':<class 'float'>,'b':(1, 10),'return':
<class 'int'>}
```

5.5.2. Обробка анотацій

Приклад 5.74.

```
def func(a: 'spam', b, c: 99):
   return a + b + c
print(func(1, 2, 3))
```

Результат: 6

```
#Вивід анотацій
print(func.__annotations__)
```

Результат:

{'a': 'spam', 'c': 99}

```
#Ітерування за анотаціями
for arg in func.__annotations__:
  print("%s : %s"% (arg,func.__annotations__[arg]))
```

Результат:

a : spam

c : 99

5.5.3. Анотації та значення за замовчуванням

Формат:

Параметр **:** анотація **=** значення

Фрагмент a: 'spam' = 4 означає, що

- аргумент a за замовчуванням отримує значення 4
- аргумент a анотований рядком 'spam':

```
def func (a:'spam'=4, b:(1, 10)=5, c:float=6) -> int:
    return a + b + c
print(func (1, 2, 3))
```

Результат: 6

Всі аргументи отримують значення за замовчуванням

```
print(func ()) # 4 + 5 + 6
```

Результат: 15

Іменовані аргументи діють як зазвичай

```
print(func (1, c = 10)) # 1 + 5 + 10
```

Результат:16

```
def func (a:'spam'=4, b:(1, 10)=5, c:float=6) -> int:
    return a + b + c
print(func (1, 2, 3))
```

Вивід словника анотацій

```
print(func.__annotations__)
```

Результат:

{ 'A': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}

Пробіли між компонентами в заголовках функцій можна використовувати або не використовувати.

Проте відмова від використання пробілів може погіршити зручність читання програмного коду.

ПІДСУМОК

Основні теми, які потрібно знати для того, щоб правильно та ефективно використовувати функції:

1. Правила задавання та передачі параметрів
2. Функції-генератори з ключовим словом **`yield`**
3. Декоратори
4. Анонімні функції **`lambda`**
5. Області видимості змінних у функціях
6. Анотації у функціях.

Контрольні запитання до розділу 5

1. Яку синтаксичну конструкцію мови Python називають функцією користувача?
2. Чим відрізняються стандартні функції від функцій користувача?
3. Запишіть функцію користувача, яка не викликає синтаксичних помилок і не виконує жодних дій.
4. Дайте визначення поняття локальних та глобальних змінних.
5. Перерахуйте стандартні атрибути об’єкта типу function.
6. Посніть причину використання необов’язкових параметрів.
7. Чим відрізняється використання позиційних параметрів від механізму зіставлення по ключах?
8. Поясніть на прикладах механізм розпакування списку або кортежу при виклику функції.
9. Які існують правила поєднання позиційних, іменованих та зіркових параметрів?
10. Дайте визначення анонімної функції та поясніть порядок її використання.

6. МОДУЛІ І ПАКЕТИ

6.1. Означення модуля

Модулем у мові Python називають будь-який файл з програмним кодом.

Кожний модуль може імпортувати інший модуль.

Після імпортування одержуємо доступ:

1. до атрибутів,
2. до змінних,
3. до функцій,
4. до класів,

які оголошені усередині імпортованого модуля.

Імпортувати можемо модулі, які написані іншими мовами

Імпортований модуль може містити програму не тільки мовою Python – так, можна імпортувати скомпільований модуль, написаний мовою С.

6.1.1. Модуль **"__main__"**

Усі програми, які ми запускали раніше, були розташовані в модулі з назвою "__main__".

Одержати ім'я модуля дозволяє визначений атрибут __name__. Виведемо назву модуля:

```
print(_name_)
```

Результат роботи: __main__

```
import mymod
print(mymod._name_)
```

Результат роботи: mymod

6.1.2. Перевірка імені модуля

Приклад 6.1. Перевірка способу запуску модуля.

```
if __name__=="__main__":
    print("Це головна програма")
 else:
    print( "Імпортований модуль")
```

```
Результат виконання: Це головна програма
```

6.1.3. Інструкція **import**

Імпортувати модуль дозволяє інструкція **import.**

Інструкція **import** має наступний формат:

import <Назва модуля1> [<as Псевдонім1>]

[,… ,<Назва модуляN> [<as ПсевдонімN>]]

1.Після ключового слова **import** вказують назву модуля.

```
import os
import testex1 as ts
import itertools as it, random as rn
```

2.Назва модуля не повинна містити розширення й шляху до файлу.

3.Назва модуля повинна повністю відповідати правилам іменувань змінних.

1.Починаємо з латинської літери,

2.Можемо включати цифри

4.Крім того, слід уникати збігу імен модулів з ключовими словами, вбудованими ідентифікаторами й назвами модулів, що входять у стандартну бібліотеку.

```
import math
import time
import mytest
```

Імпортування кількох модулів однією інструкцією **import**

За один раз можна імпортувати відразу кілька модулів, записавши їх через кому. Для прикладу підключимо модулі time і math

Приклад 6.2.

```
import time, math # Імпортуємо кілька модулів відразу
print (time.strftime ( "%d. % m. % y"))
# Поточна дата
print(math.pi) # Число pi
Результат виконання:
```

```
10. 11. 21
3.141592653589793
```

Правила доступу до атрибутів у імпортованому модулі

Після імпортування модуля його назва стає ідентифікатором.

Через ідентифікатор можна одержати доступ до атрибутів, визначених усередині модуля.

Доступ до атрибутів модуля здійснюється за допомогою точкової нотації.

Наприклад, звернутися до констант `pi` та `e`, розташованих усередині модуля `math`, можна так:

```
math.pi
math.e
```

Недолік. Якщо у модулі немає даної змінної, то виникає помилка:

```
import math
print(math.hat) # ПОМИЛКА!
```

AttributeError: module 'math' has no attribute 'hat'

6.1.4. Функція `getattr()`

Функція **`getattr()`** дозволяє одержати значення атрибута модуля по його назві, заданій у вигляді рядка.

За допомогою цієї функції можна сформувати назву атрибута динамічно під час виконання програми.

Формат функції:

getattr(<ім'я модуля>,<"Атрибут">[,<Значення за замовч.>])

a=getattr(math,"pi", 3.14)

Якщо зазначений атрибут не знайдений, виконується виключення **`AttributEerror`**. Щоб уникнути виводу повідомлення про помилку, можна в третьому параметрі вказати значення, яке буде повертатися, якщо атрибут не існує.

Приклад використання функції **`getattr`**

Приклад 6.3.

```
import math
# Атрибути pi та e існують
# Атрибут hat не існує
```

```
import math
a=["pi","e","hat"]
for i in a:
    print (i,'=',getattr (math, i))
```

Результат виконання:

```
pi = 3.141592653589793
e = 2.718281828459045
hat = Атрибуту не існує
```

6.1.5. Функція **hasattr()**

Функція **hasattr()** дозволяє перевірити існування атрибута модуля. Формат функції:

hasattr (<модуль>, <"Назва атрибута">).

Якщо атрибут існує, функція повертає значення `True`.

Приклад 6.4.

```
import math
def chattr_math(attr):
    if hasattr(math, attr):
        return "Атрибут існує"
    else:
        return "Атрибут не існує"
print (chattr_math("pi")) # Атрибут існує
print (chattr_math ("hat")) # Атрибут не існує
```

Результат виконання:

```
Атрибут існує
Атрибут не існує
```

6.1.6. Псевдонім модуля

Псевдонім задають після ключового слова **as**.

Приклад 6.5.

```
import math as m # Створення псевдоніма
print (m.pi) # Число pi
```

Результат виконання:

```
3.141592653589793
```

1.Тепер доступ до атрибутів модуля **math** може здійснюватися тільки за допомогою ідентифікатора **m**.

2.Ідентифікатор **math** у цьому випадку використовувати вже не можна.

3.Увесь уміст імпортованого модуля доступний тільки через ідентифікатор, зазначений в інструкції `import`.

Простори імен

Будь-яка глобальна змінна насправді є глобальною змінною модуля.

Тому модулі використовують як простори імен.

Якщо ваша програма складається з кількох модулів, то у кожному з модулів будуть хнаходитися свої глобальні змінні. Області видимості цих змінних не перетинаються

1. У модулі **__main__** створили змінну **myvar=10**
2. Створимо модуль з назвою **testex5.py**, у якому визначимо змінну **myvar** = 500

Області видимості цих змінних не перетинаються

Приклад 6.6.

```
import testex5  # Підключаємо файл tests.py
myvar = 10
print(testex5.myvar)
# Значення змінної myvar усередині мод.
Результат виконання: 500
print(myvar) #   Значення змінної myvar в основній програмі
Результат виконання: 10
```

Ніякого конфлікту імен немає, оскільки однойменні змінні розташовані в різних просторах імен.

6.1.7. Скомпільовані файли Python

Кожний модуль Python може бути скомпільований і перетворений в особливе внутрішнє представлення (байт-код), – це робиться для прискорення виконання коду.

Файли з відкомпільованим кодом зберігаються в папці `__pycache__`, що автоматично створюється в папці, де перебуває сам файл з початковим, невідкомпільованим кодом модуля

Скомпільовані файли мають імена виду:

`<ім'я файлу з початковим кодом>.cpython-<перші дві цифри номера версії Python>.pyc`.

Папка `__pycache__` `PyCharm` *та* `Jupyter Notebook`

За замовчуванням папка `__pycache__`

не відображається в папці проекта IDE `PyCharm`

Для відображення папки `__pycache__` необхідно

1. Вибрати в головному меню

   `File->Settings…->Editor->File Types`

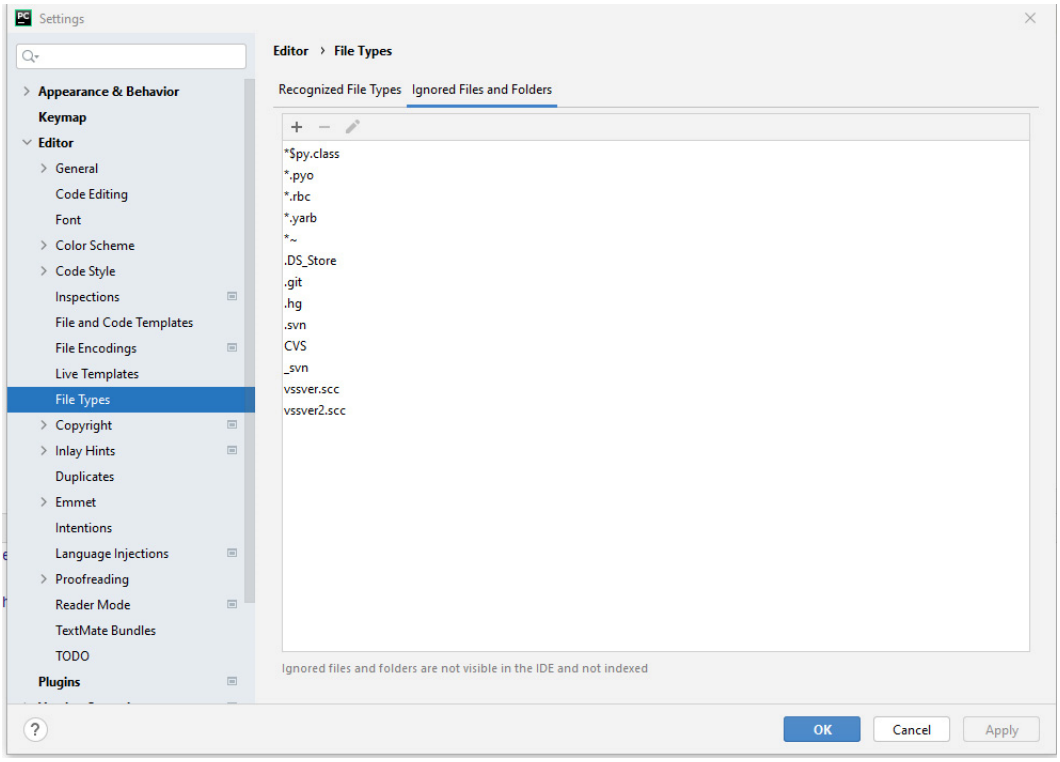
2. У вкладці з назвою «Ignore files and folders»

   видалити записи

   `*.pyc;`

   `__pycache__;`

3. Натиснути кнопку «OK»



Одержання скомпільованого файлу

Приклад 6.7.

```
import py_compile
py_compile.compile("testcom.py")
```

Таким чином, після запуску цього коду відкомпільований `testcom.py` буде збережений у файлі

`testcom.cpython-38.pyc`

у папці `__pycache__` робочого каталогу

Імпортування скомпільованих файлів

Для імпортування модуля достатньо мати тільки файл з відкомпільованим кодом

1.Скопіюємо файл `testcom.cpython-38.pyc` з папки `__pycache__` у папку з основною програмою

2. Перейменуємо його в `testcom_cpython.pyc`.

Приклад 6.8.

```
import testcom_cpython, sys
print(testcom_cpython.dc)
a=(1,2,3)
print(testcom_cpython.my_func(*a))
print(dir(testcom_cpython),'\n')
print(sorted(testcom_cpython.__dict__.keys()))
if 'testcom_cpython' in sorted(sys.modules):
    print("Модуль доступний")
```

Таким чином, щоб сховати початковий код модулів, можна поставляти програму клієнтам тільки з файлами, що мають розширення `pyc`.

6.1.8. Порядок імпортування модулів

Імпортування модуля виконується тільки при першому виклику.

Інструкції

import <module>

from <module> **import** <function>

from <module> **import** *

При кожному виклику інструкції import перевіряється наявність об'єкта модуля в словнику modules з модуля sys.

Якщо посилання на модуль перебуває в цьому словнику, то модуль повторно імпортуватися не буде.

Для прикладу виведемо ключі словника modules, попередньо відсортувавши їх.

Динамічне імпортування модулів

Інструкція **import** вимагає явної вказівки об'єкта модуля.

Щоб підключити модуль, назва якого створюється динамічно залежно від певних умов, слід скористатися функцією **__import__().**

Для прикладу підключимо модуль testex5 за допомогою функції __import__().

Приклад 6.9.

```
s = "test" + "ex5" # динамічне створення назви модуля
m = __import__(s) # Підключення модуля testex5
print(m.myvar) # Вивід значення атрибута x
Результат виконання: 500
s = "math" # динамічне створення назви модуля
m = __import__(s) # Підключення модуля math
print(m.pi) # Вивід значення атрибута pi
Результат виконання: 3.141592653589793
```

Динамічне імпортування списку модулів

Приклад 6.10.

```
my=["sys","pickle","random","math"]
mymod=[]
for i in my:
  mymod.append(__import__(i))
print(sorted(mymod[0].modules)[640:670])
print(mymod[1].dumps(my))
```

```
print(mymod[2].random())
print(mymod[3].pi)
```

Результат:

```
['sys',…, 'selectors', 'select']
b'\x80\x03]q\x00(X\x03\x00\x00\x00sysq\x01X\x06\x00\x00\x00pickl
eq\x02X\x06\x00\x00\x00randomq\x03X\x04\x00\x00\x00mathq\x04e.'
0.7301467976421705
3.141592653589793
```

6.1.9. Одержання списку ідентифікаторів модуля

Одержати список усіх ідентифікаторів усередині модуля дозволяє функція **dir().**

Крім того, можна скористатися словником **__dict__,** який містить усі ідентифікатори і їх значення.

Приклад 6.11.

```
import testcom
print(dir(testcom))
print(sorted(testcom._dict_))
print(list(testcom._dict_))
```

Результат виконання:

```
'__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'dc',
'my_func']
 ['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'dc',
'my_func']
['__name__', '__doc__', '__package__', '__loader__',
'__spec__', '__file__', '__cached__', '__builtins__', 'dc',
'my_func']
```

Інструкція **from**

Для імпортування тільки певних ідентифікаторів з модуля можна скористатися інструкцією from.

Її формат такий:

from <Назв. модуля> **import** <Ідентифікатор1> [**as** <Псевдонім1>]

[,...,<ІдентифікаторN> [**as** <Псевдонім N>]]

```
from testcom import dc as d
```

Дозволяє імпортувати модуль і зробити доступними тільки зазначені ідентифікатори.

Для довгих імен можна призначити псевдоніми, указавши їх після ключового слова as.

6.1.10. Виклик функції за псевдонімом

Як приклад зробимо доступними константу **pi** і функцію **floor()** з модуля **math**, а для назви функції створимо псевдонім.

Приклад 6.12. Інструкція from.

```
from math import pi, floor as f
print(pi) # Вивід числа pi
# Викликаємо функцію floor() через ідентифікатор f
print(f(5.49))
```

Результат виконання:

```
3.141592653589793
5
```

Ідентифікатори в круглих дужках при багаторядковому запису

Ідентифікатори можна розмістити на декількох рядках, указавши їх назви через кому усередині круглих дужок:

Приклад 6.13.

```
from math import (pi, floor,
sin, cos)
print(pi) # Вивід числа pi
print(floor(5.49))
```

```
print(round(sin(2*pi),2))
print(cos(pi))
```

6.2. Стандартні бібліотеки і сервіси

6.2.1. Огляд стандартних бібліотек

Модулі стандартної бібліотеки можна умовно розбити на групи по тематиці.

1. **Сервіси періоду виконання**. Модулі: `sys, atexit, copy, traceback, math, cmath, random, time, calendar, datetime, sets, array, struct, itertools, locale, gettext.`

2. **Підтримка циклу розробки.** Модулі: `pdb, hotshot, profile, unittest, pydoc.` Пакети `docutils, distutils.`

3. **Взаємодія з ОС** (файли, процеси). Модулі: `os, os.path, getopt, glob, popen2, shutil, select, signal, stat, tempfile.`

4. **Обробка текстів**. Модулі: `string, re, stringio, codecs, difflib, mmap, sgmllib, htmllib, htmlentitydefs.` Пакет `xml.`

5. **Багатопоточні обчислення.** Модулі: `threading, thread, queue.`

6. **Зберігання даних. Архівація.** Модулі: `pickle, shelve, anydbm, gdbm, gzip, zlib, zipfile, bz2, csv, tarfile.`

7. **Платформо-залежні модулі**. Для UNIX: `commands, pwd, grp, fcntl, resource, termios, readline, rlcompleter.` Для Windows: `msvcrt, _winreg, winsound.`

8. **Підтримка мережі. Протоколи Інтернет**. Модулі: `cgi, cookie, urllib, urlparse, httplib, smtplib, poplib, telnetlib, socket, asyncore.` Приклади серверів: `socketserver, basehttpserver, xmlrpclib, asynchat.`

9. **Підтримка Internet. Формати даних**. Модулі: `quopri, uu, base64, binhex, binascii, rfc822, mimetools, Mimewriter, multifile, mailbox.` Пакет `email.`

10. **Python про себе**. Модулі: `parser, symbol, token, keyword, inspect, tokenize, pyclbr, py_compile, compileall, dis, compiler.`

11. **Графічний інтерфейс**. Модуль `tkinter`.

6.2.2.Сервіси періоду виконання

Модуль `sys`

Модуль `sys` містить інформацію про середовище виконання програми, про інтерпретатора Python. Далі будуть представлені найбільш популярні об'єкти із цього модуля: решта можна вивчити по документації.

`exit([c])`	Вихід із програми. Можна передати числовий код завершення: 0 у випадку успішного завершення, інші числа при аварійнім завершенні програми.
`argv`	Список аргументів командного рядка. Звичайно `sys.argv[0]` містить ім'я запущеної програми, а інші параметри передаються з командного рядка.
`platform`	Платформа, на якій працює інтерпретатор.
`stdin, stdout, stderr`	Стандартний ввід, вивід, вивід помилок. Відкриті файлові об'єкти.
`version`	Версія інтерпретатора.
`setrecursionlimit(limit)`	Установка рівня максимальної вкладеності рекурсивних викликів.
`exc_info()`	Інформація про оброблюване виключення.

Приклад 6.14. Програма для задавання коду виходу

```
import sys
while True:
   a=input("Ведіть код виходу")
   if a.isdigit: sys.exit(int(a))
```

Результат роботи:

```
Ведіть код виходу 25
Process finished with exit code 25
```

Приклад 6.15.

```
import sys
print("Ім'я програми:",sys.argv[0])
print("Платформа:",sys.platform)
```

Результат роботи:

```
Ім'я програми: C:/NPMData/Service.py
Платформа: win32
```

Приклад 6.16.

```
import sys
print("Версія інтерпретатора:\n",sys.version)
sys.exit(24)
```

Результат роботи:

```
Версія інтерпретатора:
3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit
(AMD64)]
SystemExit: 24
```

6.2.3. Модуль `time` (1.1.1970)

Цей модуль містить функції для одержання поточного часу й перетворення форматів часу.

time.altzone - зсув DST (*Daylight Saving Time*) часового поясу в секундах на захід від нульового меридіана. Якщо годинний пояс перебуває на сході, зсув негативний.

Приклад 6.17.

```
import time
print("Часовий зсув:", time.altzone, "сек" )
```

Результат роботи:

```
Часовий зсув: -10800 сек
```

time.asctime([t]) **-** перетворює кортеж або struct_time у рядок виду "День Місяць Число Час Рік". Якщо аргумент не зазначений, використовується поточний час.

Приклад 6.18.

```
import time
t = time.localtime()
print (time.asctime(t))
```

Результат роботи:

```
Wed Nov 10 14:29:07 2021
```

time.ctime([сек]) – перетворить час, виражений в секундах з початку епохи, на рядок виду "День Місяць Число Час Рік".

Приклад 6.19.

```
import time
print (time.ctime())
```

Результат роботи:

```
Wed Nov 10 14:29:07 2021
```

time.sleep(сек) – призупинити виконання програми на задану кількість секунд.

Приклад 6.20.

```
import time
print ("Перед сном",time.ctime())
time.sleep(20)
print ("Після  сну",time.ctime())
```

Результат роботи:

Перед сном Fri Nov 20 16:48:29 2020

Після сну Fri Nov 20 16:48:49 2020

time.strftime(формат, [t]) – перетворить кортеж або `struct_time` у рядок по формату:

ФОРМАТ	ЗНАЧЕННЯ
% a	Скорочена назва дня тижня
% A	Повна назва дня тижня
% b	Скорочена назва місяця
% B	Повна назва місяця
% c	Дата й час
% d	День місяця [01,31]

% H	Година (24-годинний формат) [00,23]
% I	Година (12-годинний формат) [01,12]
% j	День року [001,366]
% m	Номер місяця [01,12]
% M	Число хвилин [00,59]
% p	До полудня або після (при 12-годинному форматі)
% S	Число секунд [00,61]
% U	Номер тижня в році (нульовий тиждень починається з неділі) [00,53]
% w	Номер дня тижня [0(Sunday),6]
% W	Номер тижня в році (нульовий тиждень починається з понеділка) [00,53]
% x	Дата
% X	Час
% y	Рік без століття [00,99]
% Y	Рік зі століттям
% Z	Тимчасова зона
% %	Знак '%'

Приклад 6.21.

```
import time
s1 = time.strftime("%A, %D %B %Y %H:%M:%S + 0000",
time.gmtime())
print("GMT", s1)
s2 = time.strftime("%c")
print("Date Time", s2)
s3 = time.strftime(" %R ")
print("Only Time", s3)
```

Результат роботи:

```
GMT Wednesday, 11/10/21 November 2021 12:37:32 + 0000
Date Time Wed Nov 10 14:37:32 2021
```

```
Only Time  14:37
```

time.time() - час, виражений в секундах з початку епохи.

time.timezone - зсув місцевого годинного пояса в секундах на захід від нульового меридіана. Якщо годинний пояс перебуває на схід, зсув негативний.

Приклад 6.22.

```
import time
print("Час з початку епохи:",time.time())
print("Зсув часу в секундах:",time.timezone)
```

Результат роботи:

```
Час з початку епохи: 1605883931.2408626
Зсув часу в секундах: -7200
```

6.2.4. Модулі `array` і `struct`

Ці модулі реалізують низькорівневий масив і структуру даних. Основне їхнє призначення - розбір двійкових форматів даних.

Модуль **`itertools`**

Цей модуль містить набір функцій для роботи з ітераторами. Ітератори дозволяють працювати з даними послідовно, як ніби вони виходили в циклі. Альтернативний підхід – використання списків для зберігання проміжних результатів – вимагає часом великої кількості пам'яті, тоді як використання ітераторів дозволяє одержувати значення на момент, коли вони дійсно потрібні для подальших обчислень.

Модуль **`gettext`**

При інтернаціоналізації програми важливо не тільки передбачити можливість використання декількох культурних середовищ, але й переклад повідомлень і меню програми на відповідну мову. Модуль gettext дозволяє спростити цей процес досить стандартним способом. Основні повідомлення програми пишуться англійською мовою.

Переклади рядків, відзначених у програмі спеціальним образом, даються у вигляді окремих файлів, по одному на кожну мову (або культурне середовище). Уточнити нюанси використання `gettext` можна по документації до `Python`.

6.2.5. Шляхи завантаження модулів

Завантаження **from** *з використанням* **(*)**

Формат інструкції:

from <Назва модуля> import *

from дозволяє імпортувати з модуля всі ідентифікатори. Для прикладу імпортуємо всі ідентифікатори з модуля **math**.

Приклад 6.23. Імпорт усіх ідентифікаторів з модуля.

```
# Імпортуємо всі ідентифікатори з модуля math
from math import *
print (pi) # Вивід числа pi
print(floor(5.49)) # Викликаємо функцію floor ()
```

Результат виконання:

```
3.141592653589793
5
```

Недоліки повного імпортування

Необхідно враховувати, що імпортування всіх ідентифікаторів з модуля може порушити простір імен головної програми.

Причина

1.У різних модулях можуть міститися ідентифікатори з однаковими назвами.

2. Ідентифікатори, що мають однакові імена з головною програмою, будуть перезаписані.

Приклад перезапису ідентифікаторів

Створимо два модулі й підключимо їх за допомогою інструкцій **from** і **import**.

Приклад 6.24.

```
# Вміст файлу module1.py s = "Значення з модуля module1"
# Вміст файлу module2.py s = "Значення з модуля module2"
# початковий код основної програми
from module1 import *
from module2 import *
```

```
print(s)  # Виведе: "Значення з модуля module2"
import module1, module2
print(module1.s) # Виведе: "Значення з модуля module1"
print(module2.s) # Виведе: "Значення з модуля module2"
```

```
Значення з модуля module2
Значення з модуля module1
Значення з модуля module2
```

Шляхи пошуку модулів

Дотепер ми розміщали модулі в одній папці з файлом основної програми. У цьому випадку немає необхідності задавати шляхи пошуку модулів, тому що папка з файлом, що виконується, автоматично додається в початок списку шляхів.

```
import sys
```

Список **sys.path** містить шляхи пошуку, які одержують за замовчуванням: шляхи до доступних каталогів, які містять робочий каталог та шляхи до системних бібліотек;

У список **sys.path** можна додати додаткові шляхи за бажанням програміста.

Існує два способи додавання шляхів:

1.Статичний спосіб

2. Динамічний спосіб

Цей код дозволяє вивести поточний список шляхів

Приклад 6.25.

```
import sys
s="\n".join(sys.path)
print(s)
```

Результат:

```
F:\Notebooks
C:\MYFOLDER
C:\Anaconda3\python38.zip
C:\Anaconda3\DLLs
C:\Anaconda3\lib
C:\Anaconda3
```

```
C:\Anaconda3\lib\site-packages
C:\Anaconda3\lib\site-packages\locket-0.2.1-py3.8.egg
C:\DYNFOLDER
C:\Anaconda3\lib\site-packages\win32
C:\Anaconda3\lib\site-packages\win32\lib
C:\Anaconda3\lib\site-packages\Pythonwin
C:\Anaconda3\lib\site-packages\IPython\extensions
C:\Users\Novotarskyi\.ipython
```

Модифікація змінної PYTHONPATH

1.Для додавання змінної в меню **Пуск** вибираємо пункт **Панель керування** (або Налаштування І Панель керування).

2.У вікні, що відкрилося, вибираємо пункт Система.

3.Клацаємо на посиланні Додаткові параметри системи. 4.Переходимо на вкладку Додатково й натискаємо кнопку Змінні середовища.

5.У розділі Змінні середовища користувача натискаємо кнопку Створити.

6.У поле Ім'я змінної вводимо PYTHONPATH, а в полі Значення змінної задаємо шляхом до папки – наприклад, **`C:\MYFOLDER`**

Файли з розширенням **`*.pth`**

Для задавання нових шляхів до файлів необхідно створити файл з розширенням *.pth і розмістити його в каталозі:

Lib\site-packages*.pth

1.Назви таких файлів можуть бути довільними, головне, щоб вони мали розширення **`pth.`**

2.Кожний шлях повинен бути розташований на окремому рядку.

3.Назву диску писати з великої літери

Наприклад:

C:\DYNFOLDER

Приклад 6.26.

1. Створіть файл `mypath.pth` у каталозі

External Libreries\Lib\site-packages\mypath.pth (PyCharm)

Anaconda3\Lib\ site-packages\mypath.pth (Jupyter)

з наступним вмістом:C:\DYNFOLDER

2. У папку папку C:\MYFOLDER

запишіть файл mymod.py з наступним вмістом:

s='Ми отримали доступ до модуля+ _name_ + з папки MYFOLDER'

3.У папку папку C:\DYNFOLDER

запишіть файл distmod.py з наступним вмістом:

s='Ми отримали доступ до модуля+ _name_ + з папки DYNFOLDER'

Виконайте код з головного модуля

```
import mymod
import distmod
s="Ми отримали доступ до модуля" + __name__
print(s)
print(mymod.s)
print(distmod.s)
```

Правила роботи з файлом **mypath.pth**

1.Каталоги повинні існувати, а якщо ні, то вони не будуть додані в список **sys.path**.

Приклад 6.27.

а)В файл Anaconda3\Lib\ site-packages\mypath.pth записуємо

рядок C:/NOFOLDER

б) Перевіряємо вміст списку sys.path

```
import sys
s="\n".join(sys.path)
print(s)
```

Результат:

```
F:\Notebooks
C:\MYFOLDER
…
C:\DYNFOLDER
```

2.При пошуку модуля список **sys.path**, проглядається зліва направо. Пошук припиняється після першого знайденого модуля. Таким чином, якщо в папці C:\MYFOLDER і C:\DYNFOLDER існують однойменні модулі, то буде використовуватися модуль із папки, яка стоїть перед даною папкою.

Приклад 6.28.

```
import dblmod
print(dblmod.s)
```

Результат

```
Завантажено модуль з папаки MYFOLDER
```

4.Список **sys.path** можна змінювати із програми за допомогою списковых методів.

Наприклад, додати каталог у кінець списку можна за допомогою методу append(), а в його початок – за допомогою методу insert().

Приклад 6.29. Зміна списку шляхів пошуку модулів

```
import sys
sys.path.append(r"C:\APFOLDER")
# Додаємо в кінець списку
sys.path.insert(0, r"C:\INSFOLDER")
# Додаємо в початок списку
print("\n".join(sys.path))
```

Результати виконання:

```
C:\INSFOLDER
C:\APFOLDER
```

У цьому прикладі

Додали папку C:\INSFOLDER у початок списку.

Додали папку C:\APFOLDER у кінець списку.

ПАКЕТИ

Означення пакета

Пакетом називають папку з модулями, у якій розташований файл ініціалізації __init__.py.

Файл ініціалізації може бути порожнім або містити код, який буде виконаний при першому доступі до пакета.

У будь-якому разі він обов'язково повинен бути присутнім у папці з модулями.

mypack

__init__.py

subsamp.py

subpack

Приклад структури файлів і каталогів:

Notebook

note20.ipynb # Основний файл з програмою

mypack# Папка на одному рівні вкладеності з first.py

__init__.py # Файл ініціалізації

sample1.py # Модуль mypack\sample1.py

subpack # Вкладений пакет

__init__.py # Файл ініціалізації

subsamp1.py

subsamp2.py

Вміст файлу **__init__.py** для пакету **mypack** наведений в наступному прикладі.

Приклад 6.30.

```
import mypack as m
print("__init__",__name__)
```

Заповнимо модулі sample1.py, subsamp1.py і subsamp2.py як показано в прикладі 3

Вміст модулів sample1.py, subsamp1.py, subsamp2.py

```
msg = "Модуль {0:50}".format(__name__)
```

Імпорт модулів в основний файл

Приклад 6.31.

```
print("Доступ до модуля sample1")
import mypack.sample1 as s1
print(s1.msg)
print(s1.__name__)
```

Результат:

```
Доступ до модуля sample1
Модуль mypack.sample1
mypack.sample1
```

Приклад 6.32.

```
print("\nДоступ до модуля subsump1")
import mypack.subpack.subsamp1 as ss1
print(ss1.msg)
print(ss1.__name__)
```

Результат:

```
Доступ до модуля subsamp1
Модуль mypack.subpack.subsamp1
mypack.subpack.subsamp1
```

Приклад 6.33.

```
print("\nДоступ до модуля subsump2")
import mypack.subpack.subsamp2 as ss2
print(ss2.msg)
print(ss2.__name__)
```

Результат:

```
Доступ до модуля subsamp2
Модуль mypack.subpack.subsamp2
mypack.subpack.subsamp2
```

6.2.6. Порядок доступу до модулів пакета

Як видно з прикладу, пакети дозволяють розподілити модулі по каталогах.

Щоб імпортувати модуль, розташований у вкладеному каталозі, необхідно вказати шлях до нього, перелічивши імена каталогів через крапку.

Якщо модуль розташований у каталозі C:\Notebook\mypack\subpack\, то шлях до нього повинен бути записаний так: **mypack.subpack** за умови, що __main__ знаходиться в одному каталозі з **mypack**.

Застосування інструкції **import**

При використанні інструкції `import` шлях до модуля повинен включати не тільки назви каталогів, але й назву модуля без розширення:

```
import mypack.subpack.subsamp1
```

Одержати доступ до ідентифікаторів всередині імпортованого модуля можна в такий спосіб:

Приклад 6.34.

```
import mypack.subpack.subsamp1
print(mypack.subpack.subsamp1.msg)
```

Результат роботи:

```
Модуль mypack.subpack.subsamp1
```

Псевдоніми

При використанні довгих ідентифікаторів виникають незручності. Можна створити псевдонім, указавши його після ключового слова as, і звертатися до ідентифікаторів модуля через нього:

Приклад 6.35.

```
import mypack.subpack.subsamp1 as ss1
print(ss1.msg)
```

Результат роботи:

```
Модуль mypack.subpack.subsamp1
```

Застосування інструкції **from**

При використанні інструкції **from** існує два способи завантаження:

1. імпортування модуля,
2. імпортування **визначених ідентифікаторів** з модуля.

Щоб імпортувати об'єкт модуля, його назву слід указати після ключового слова **import**:

Приклад 6.36.

```
from mypack.subpack import subsamp1
print(subsamp1.msg)
```

Результат роботи:

```
Модуль mypack.subpack.subsamp1
```

Імпортування визначених ідентифікаторів

Для імпортування тільки визначених ідентифікаторів назва модуля вказується в складі шляху, а після ключового слова import через кому перелічуються ідентифікатори.

Додамо в модуль subsamp1 :

```
msg1 = "Це msg1 з модуля {0}".format(__name__)
msg2 = "Це msg2 з модуля {0}".format(__name__)
```

Приклад 6.37

```
from mypack.subpack.subsamp1 import msg, msg1,msg2
print (msg)
print (msg1)
print (msg2)
```

Результат роботи:

```
Модуль mypack.subpack.subsamp1
Це msg1 з модуля mypack.subpack.subsamp1
Це msg2 з модуля mypack.subpack.subsamp1
```

Імпортування всіх ідентифікаторів

Якщо необхідно імпортувати всі ідентифікатори з модуля, то після ключового слова import вказують символ *:

Приклад 6.38.

```
from mypack.subpack.subsamp1 import *
print (msg)
print (msg1)
print (msg2)
```

Результат роботи:

```
Модуль mypack.subpack.subsamp1
Це msg1 з модуля mypack.subpack.subsamp1
Це msg2 з модуля mypack.subpack.subsamp1
```

Імпортування кількох модулів одночасно

Інструкція from дозволяє також імпортувати відразу кілька модулів з пакета.

Для цього всередині файлу ініціалізації __init__.py в атрибуті __all__ необхідно вказати список модулів, які будуть імпортуватися за допомогою виразу:

from шлях до пакету import *

Змінимо вміст файлу mypack.subpack._init_.py :

__all__ = ["subsamp1", " subsamp2"]

Приклад 6.39.

```
from mypack.subpack import *
print(subsamp1.msg)
print(subsamp1.msg1)
print(subsamp1.msg2)
print(subsamp2.msg)
```

Результат роботи:

```
Модуль mypack.subpack1.example1
Це msg1 з модуля mypack.subpack1.subsamp1
Це msg2 з модуля mypack.subpack11.subsamp1
Модуль mypack.subpack1.subsamp2
```

Після ключового слова **from** вказується лише шлях до каталогу без імені модуля. У результаті виконання інструкції from усі модулі, зазначені в списку __all__, будуть імпортовані в простір імен модуля __main__

мпортування модулів всередині пакета

Інструкція **from** підтримує відносний імпорт модулів. Щоб імпортувати модуль, розташований у тому ж каталозі, перед назвою модуля вказують крапку:

from .<ім'я модуля> **import** *

Щоб імпортувати модуль, розташований у батьківському каталозі, перед назвою модуля вказують дві крапки: **from** ..<ім'я модуля> **import** *

Якщо необхідно звернутися ще рівнем вище, то вказують три крапки:

from ... <ім'я модуля> **import** *

Чим вище рівень, тим більше крапок необхідно вказати. Після ключового слова from можна вказувати одні тільки крапки – у цьому випадку ім'я модуля вводиться після ключового слова import.

from . . import <ім'я модуля>

Приклади відносного імпорту модулів

Розглянемо відносний імпорт на прикладі. Для цього створимо модуль mypack.subpack.relative.py, як показано в прикладі

Вміст модуля relative.py

```
# Імпорт модуля subsamp1 з поточного каталогу
from .import subsamp1 as ss1
var1 ="Значення з: {0}".format(ss1.msg)
from  .subsamp1 import  msg as ms1
var2 = "Значення з: {0}".format(ms1)
#Імпорт модуля sample1 з батьківського каталогу
from ..import sample1 as smp
var3 = "Значення з: {0}".format(smp.msg)
from  ..sample1 import msg as m
var4 = "Значення з: {0}".format(m)
```

Тепер розглянемо вміст основного файлу `four.py`

і запустимо його.

Приклад 6.40. Вміст файлу **`four.py`**

```
from mypack.subpack import relative as ss2
print(ss2.var1)
print(ss2.var2)
print(ss2.var3)
print(ss2.var4)
Результат роботи:
Значення з: Модуль mypack.subpack.subsamp1
Значення з: Модуль mypack.subpack.subsamp1
Значення з: Модуль mypack.sample1
Значення з: Модуль mypack.sample1
```

Цей пакет і набір утиліт поки що не входить у стандартну поставку Python, однак про нього потрібно знати тим, хто прагне швидко готовити документацію (посібники користувача) для своїх модулів.

Цей пакет використовує спеціальну мову розмітки (Restructured text (RST)), з якого потім легко виходить документація у вигляді HTML, Latex і в інших форматах. Текст у форматі RST легко читати й у початковому виді. Із цим інструментом можна познайомитися на http://docutils.sourceforge.net

6.2.7. Пакет `distutils`

Даний пакет надає стандартний шлях для поширення власних **Python-дистрибутивів**.

1.Створити в проекті окрему папку, наприклад mydist.

2. Переписати в цю папку файли вашого проекту та створені вами пакети, що необхідні для роботи проекту.

3. Створити файл документації Documentation.txt

4.Створити файл інструкції README.txt.

5.Створити файл setup.py

6. Перейти в папку проекту.

7. В командному рядку Windows набрати:

```
python setup.py sdist
```

Приклад 6.41. Приклад створення папки дистрибутиву.

```
mydist
    README.txt
    Documentation.txt
    first.py
    mypack
        __init__.py
        sample1.py
        subpack
            __init__.py
            subsamp1.py
```

Приклад 6.42. Створення файлу `setup.py`

В папці проекту створюємо файл `setup.py` з таким вмістом:

```
#setup.py
from distutils.core import  setup
```

```
setup(name =
 "mydist",
 version =1.0,
 py_modules=["first"],
 packages=["mypack","mypack.subpack"])
```

Приклад 6.43. Створення файлу дистрибутиву

1.Відкриваємо консоль Windows

2. Переходимо в папку проекту

3. Виконуємо командний рядок :

```
python setup.py sdist
```

4. В папці проекту буде створена папка `dist`.

5. В папці знаходитиметься дистрибутив з розширенням

```
*.tar.gz
```

Контрольні запитання до розділу 6.

1. Дайте визначення модуля у мові Python.
2. Як перевірити ім’я поточного модуля, у якому розміщується код?
3. Опишіть можливі варіанти статичного та динамічного імпортування модулів.
4. Які функції використовують до зчитування та модифікації атрибутів модуля?
5. Як отримати скомпільований файл у мові Python?
6. Які стандартні бібліотеки мови Python вам відомі?
7. Опишіть призначення та спосіб модифікації глобальної змінної PYTHONPATH.
8. Що називають пакетом у мові Python?
9. Який порядок доступу до окремих модулів пакету?
10. Опишіть порядок застосування пакету distutils для формування дистрибутивів.

7. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

7.1. Основні поняття ООП

7.1.1. Парадигми програмування

Імперати́вне (процедурне) програмува́ння — парадигма програмування. Процес отримання результатів відбувається в результаті виконання послідовності інструкцій.

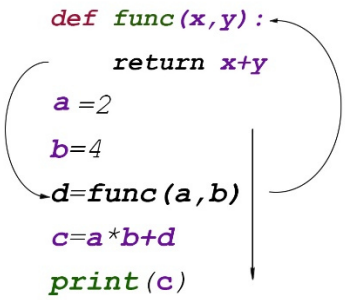


Рис. 7.1. Приклад імперативного коду

Імперативні мови програмування відрізняються від мов функціонального програмування(Scala, Haskell) і логічного програмування (Prolog).

Функціональне програмування

Функціональне програмування — парадигма програмування, яка розглядає програму як обчислення математичних функцій та уникає зміни станів та змінних даних.

Функціональне програмування – застосування функцій,

Іншими словами, *функціональне програмування* є способом створення програм, в яких єдиною дією є виклик функції, єдиним способом розбиття програми є створення нового імені функції та задання для цього імені виразу, що обчислює значення функції, а єдиним правилом композиції є оператор суперпозиції функцій.

Об'єктно-орієнтоване програмування, або ООП - це парадигма програмування, яка забезпечує засіб структурування програм так, що специфічні *властивості* та *поведінка* об’єднані в окремих об'єктах.

ООП дозволяє:

-розділити програму на фрагменти

-описати предмети реального світу у вигляді об'єктів,

-організувати зв'язки між цими об'єктами.

ООП - підхід для моделювання конкретних речей,

-таких як автомобілі,

-зв’язків між речами: компанії та співробітники, студенти та викладачі тощо.

7.1.2. Структура ООП

Описуємо реальні об'єкти як об'єкти програмного забезпечення, які мають деякі дані, що визначають їх стани, і можуть виконувати певні дії, які представлені функціями.

Структура ООП, яка вперше була запропонована доктором комп’ютерних наук, страшим архітектором по розробці програмного забезпечення Sun та IBM Річардом Гебріелом.

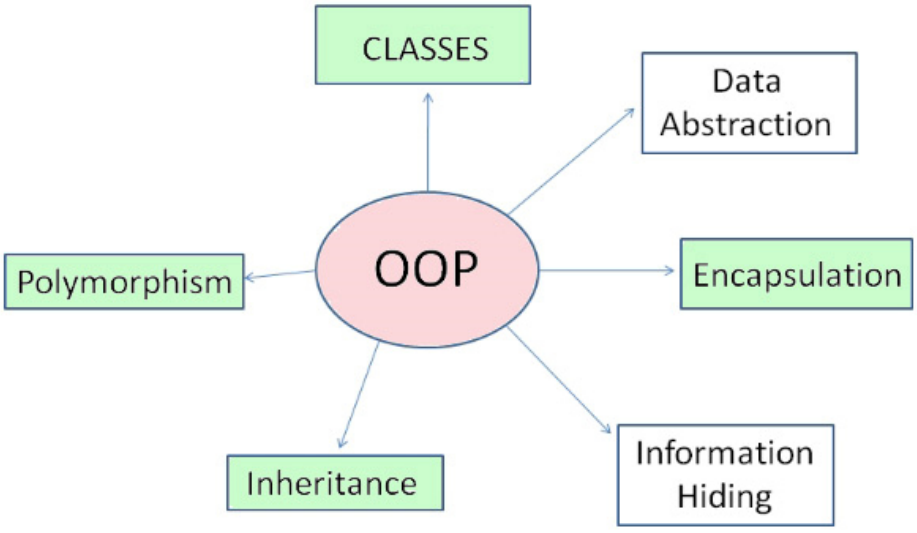


Рис. 7.2. Загальна структура ООП

ООП ґрунтується на основних концепціях розробки: абстрагування, інкапсуляція, спадкування та поліморфізм.

Мова Python - об'єктно-орієнтована мова програмування. Це означає, що Python побудований з урахуванням наступних принципів:

1. Усі дані в ньому представлені як об'єкти.

2. Програму можна створити як набір взаємодіючих об'єктів, що посилають один одному повідомлення.
3. Кожний об'єкт має власну частину пам'яті і може складатися з інших об'єктів.
4. Кожний об'єкт має тип.
5. Усі об'єкти одного типу можуть приймати ті самі повідомлення (і виконувати ті самі дії).

Основні поняття

При процедурному програмуванні програму розбивають на частини відповідно до алгоритму: кожна частина є складовою частиною одного алгоритму.

При об'єктно-орієнтованому програмуванні програму будують як сукупність взаємодіючих об'єктів.

Об'єкт - це щось, що має

значення (стан),

тип (поведінка)

індивідуальність (ім'я).

Визначити об'єкт в предметній області – це абстрагуватися від більшості його властивостей, *концентруючись на істотних для задачі властивостях.*

7.1.3. Стан і поведінка об'єкта

Об'єкти створюють на основі **класів**. **Клас** є шаблоном об'єкта !!!!

Усі об'єкти, які визначені користувачем, є екземплярами класу. Об'єкт, створений на основі деякого класу, називають **екземпляром класу**.

Кожний об'єкт зберігає свій **стан, який визначається атрибутами.** Об'єкт також має певний набір **методів**. Методи визначають **поведінку** об'єкта.

Об'єкти одного і того ж класу мають спільну поведінку.

Можуть одночасно існувати кілька екземплярів класу, створених на основі одного і того ж класу.

7.1.4. Формальні визначення

Об'єкт - це будь-яка сутність в Python

(число, рядок, список, кортеж, словник, функція, тобто ВСЕ).

```
123, "string", [1,2,3,4], (5,6,7),{"a":1,"b":2},
print(type(123))
<class 'int'>
```

Клас це шаблон об’єктів. Тому типом класу є тип type:

```
class MyClass: pass
    #атрибути
    #методи
print(type(MyClass))
Результат: <class 'type'>
```

Екземпляр

```
mc = MyClass()  #задаємо екземпляр класу
print(type(mc))
Результат: <class '__main__.MyClass'>
```

Визначення класу

У мові Python для визначення класу використовується оператор class. Клас описують за допомогою ключового слова **class** за наступною схемою:

```
class <Назва класу>[(<Клас1>[, ... , <Класn>])]:
    [""" Рядок документування """]
    <Опис атрибутів>
    <Опис методів>
```

class оператор класу

Назва класу повинна повністю відповідати правилам іменування змінних. Круглі дужки після назви і двокрапка - ():

Після назви класу в круглих дужках можна вказати один або кілька базових класів через кому. При відсутності базових класів дужки не вказують.

<Опис атрибутів> Змінні даного класу.

<Опис методів> Функції даного класу.

Функції мають обов’язковий параметр **self**

Отже клас визначає **тип** об'єкта, тобто його можливі стани й набір операцій.

Виконання інструкцій класу

Всі вирази всередині інструкції **class** виконуються при створенні класу, а не його екземпляра.

Приклад 7.1. Створення класу

```
class MyClass:

    """ Це рядок документування """
    a=20+10
    print("Інструкції виконуються відразу")
```

Цей приклад містить лише визначення класу **MyClass** і не створює екземпляр класу.

Як тільки потік виконання досягне інструкції **class**, повідомлення, зазначене у функції **print**(), буде відразу виведене.

7.1.5. Створення атрибутів і методів класу

Створення атрибута класу аналогічно створенню звичайної змінної.

Створення метода класу. Метод всередині класу створюється так само, як і звичайна функція, за допомогою інструкції **def.**

У методах класу перший параметр обов'язково слід указати явно, він автоматично передає посилання на екземпляр класу.

Загальноприйнято цей параметр називати ім’ям **self**, хоча це й не обов'язково.

Доступ до атрибутів і методів класу всередині методу проводиться через змінну `self` за допомогою точкової нотації. до атрибута `x` з методу класу можна звернутися так: `self.x.`

Створення екземпляру класу

Щоб використовувати атрибути й методи класу, необхідно створити екземпляр класу згідно з наступним синтаксисом:

<Екземпляр класу> = <Назва класу>(<Параметри>])

Наприклад:

```
class MyClass:
    """Рядок документування"""
    # Атрибути класу
    # Методи класу
a=MyClass()
```

Отже, **MyClass** – це назва класу (**велика буква**)

a – назва об'єкта- екземпляру класу (**маленька буква**)

Доступ до методів класу

Доступ до методів класу використовує такий формат:

<**Екземпляр класу**>.<**Ім'я методу**>([<**Параметри**>])

Наприклад: a.mymethod()

```
class MyClass:                          def
mymethod(self,x):
        print("Attribute x =",x)
c = MyClass ()
d = MyClass ()
c.mymethod(2)
d.mymethod(3)
```

Результат:

```
Attribute x = 2
Attribute x = 3
```

Доступ до атрибутів класу

Доступ до атрибутів класу здійснюється аналогічно:

<**Екземпляр класу**>.<**Ім'я атрибута**>

Наприклад: **a.myattribute**

```
class MyClass:
    """Доступ до атрибутів"""
    myattribute=128
```

```
    def mymethod(self):
        print("Внутрішній доступ", self.myattribute)
c = MyClass()
d = MyClass()
c.mymethod()
k=d.myattribute
print("Зовнішній доступ:",k )
```

Посилання на екземпляр класу при доступі до атрибутів та методів класу інтерпретатор передає автоматично.

Приклад задавання класу

1.Визначимо клас `MyClass` з атрибутми `x,y,z` і методом `adder(),` який повертає суму атрибутів.

2. Створимо екземпляр класу й викличемо метод.

Приклад 7.2.

```
class MyClass:
   x, y, z = 10, 20, 30
   def adder(self):# self-це посилання на екз. класу
      return self.x+self.y+self.z
c = MyClass () # Створення екземпляра класу
s=c.adder()  # self не вказують при виклику методу
print("{0}{1:+d}{2:+d}={3}".format(c.x,c.y,c.z,s))
```

Результат: 10+20+30=60

7.1.6. Конструктор класу: метод `__init__()`

При створенні екземпляра класу інтерпретатор автоматично викликає метод ініціалізації **`__init__()`**. В інших мовах програмування такий метод прийнято називати конструктором класу.

Формат методу:

```
def__init__(self[,<Знач1>[,... , <Значn>]]):
  <Інструкції>
```

За допомогою методу `__init__()` можна присвоїти початкові значення атрибутам класу. При створенні екземпляра класу параметри цього методу вказують після імені класу в круглих дужках:

<Екз.класу>=<Ім'я класу>([<Знач1>[,...,<Значn>]])

Приклад використання методу **`__init__()`:**

Приклад 7.3.

```
#Створюємо клас
class MyClass:
    def __init__(self, value1, value2):#Конструктор
        self.x = value1 # Атрибути екземпляра
        self.y = value2
        print("Спрацював конструктор")
    print("Створено клас MyClass")
# Створюємо екземпляр класу
c = MyClass(100, 300)
print("Створено екземпляр класу MyClass")
print(c.x, c.y)
```

Результат виконання:

```
Створено клас MyClass
Спрацював конструктор
Створено екземпляр класу MyClass
100 300
```

Приклад класу **`Person`** *з інформацією про людину*

Приклад 7.4.

```
class Person:
    def __init__(self, name, job, pay):
        # Конструктор вимагає трьох аргументів
        self.name = name
        # Атрибути потрібно заповнити при створенні екземпляра
        self.job = job  # self – це новий екземпляр класу
```

```
        self.pay = pay
petro = Person(" Petrenko Petro","Python developer", 10000)
print(petro.name,"\n", petro.job,"\n", petro.pay)
```

Ми бачимо, що екземпляр класу **petro** був успішно створений, оскільки всі параметри для методу **__init__** задані коректно.

Зробимо спробу створити екземпляр класу `Person` з неповною кількістю параметрів.

```
#Не заповнили job і name
ivan = Person("Ivanov Ivan")
print(ivan.name)
```

Результат виконання:

```
Traceback (most recent call last):
0: __init__() missing 2 required positional arguments:
'job' and 'pay'
```

При створенні екземпляра класу `ivan` виникло виключення **TypeError**, оскільки не всі обов'язкові параметри були задані на етапі ініціалізації.

Для того, щоб уникнути таких помилок, деяким параметрам методу **__init__** можна задати значення за замовчуванням:

Приклад 7.5. Person з параметрами за замовчуванням

```
class Person:
    def __init__(self, exs_name, exs_job = None, exs_pay = 0):
        # Конструктор не вимагає трьох аргументів
        self.name = exs_name
        self.job =  exs_job
        self.pay =  exs_pay
petro = Person("Petrenko Petro","Python developer", 10000 )
print(petro.name, petro.job, petro.pay)
ivan = Person("Ivanov Ivan")
print(ivan.name, ivan.job, ivan.pay)
```

Результат виконання:

```
Petrenko Petro Python developer 10000
Ivanov Ivan None 0
```

7.1.7. Деструктор класу: метод **`__del__()`**

Перед знищенням об'єкта (екземпляра класу) автоматично викликається метод, який називають деструктором. У мові **Python** деструктор реалізується у вигляді визначеного методу **`__del__()`.**

Метод не буде викликатися, якщо на екземпляр класу існує хоча б одне посилання. Інтерпретатор самостійно опікується видаленням об'єктів.

Тому використання деструктора в мові **Python** необхідно тільки в тому випадку, коли Ви плануєте спеціальний порядок видалення об'єкта

(наприклад, збереження значень його атрибутів).

Приклад 7.6. Явний виклик деструктора

```
class MyClass:
   def __init__(self,x):
      print( "Виконано метод __init__() ")
      self.x=x
      print(self.x)
   def __del__(self):   # Деструктор класу
      self.x=0
      print("Змінили значення self.x=",self.x)
c1 = MyClass(128)
del c1
```

Результат:

```
Виконано метод __init__()
128
Змінили значення self.x= 0
```

Приклад 7.7. `__del__` не викликається при існуванні додаткового

посилання на об’єкт

```
class MyClass:
```

```
    def __init__(self):
        print( "Виконано метод __init__() ")
    def __del__(self):   # Деструктор класу
        print("Виконано метод __del__()")
c1 = MyClass()
del c1
c2 = MyClass()
c3 = c2 # Створюємо посилання на екземпляр класу
del c2 #Нічого не виведе, оскільки існує посилання
print("C2 не видалено. Оскільки є C3")
del c3       #Результат
Виконано метод __init__()
Виконано метод __del__()
Виконано метод __init__()
C2 не видалено. Оскільки є C3
Виконано метод __del__()
```

7.1.8. Приклади створення класів

Уявіть, що ми працюємо в компанії "Тесла" Алана Маска. Ми отримали завдання написати програму для автомобіля, яка зможе спілкуватися з водієм.

Зараз ми розглянемо найпростіший варіант такої програми. Для цього ми створюємо клас Car, який включає доступні атрибути та методи. Атрибути класу включають назва, виробник та модель.

Методи класів імітують режими запуску та зупинки двигуна.

Під час ініціалізації об’єкта **mycar** ми створили атрибут об’єкта **inner**, який зберігає поточний стан автомобіля.У початковому стані двигун вимкнений. Отже, значення внутрішнього атрибута **inner** дорівнює **False**. Виклик відповідних методів дозволяє нам контролювати параметри автомобіля. Створюємо клас для опису автомобіля

```
class Car:
    # Створюємо атрибути класу
```

```
    name = "s200"
    made= "mercedez"
    model= 2005
    # Створюємо методи класу
    def __init__(self, state=False):
         self.inner=state
    def start(self):
        if not self.inner:
            self.inner= not self.inner
            print("Запускаю двигун")
        else: print("Двигун вже працює")
    def stop(self):
        if self.inner:
            self.inner = not self.inner
            print("Зупиняю двигун")
        else: print("Двигун вже зупинено")
mycar=Car()
mycar.start()
mycar.start()
mycar.stop()
mycar.stop()
mycar.start()
mycar.stop()
print("Назва:",mycar.name)
mycar.model=2010
print("Модель:",mycar.model)
```

Результат:

```
Запускаю двигун
Двигун вже працює
```

```
Зупиняю двигун
Двигун вже зупинено
Запускаю двигун
Зупиняю двигун
Назва: s200
Модель: 2010
```

Приклад програми

Ця програма імітує взаємодію між викладачем та його учнями. Клас **Teacher** описує дії вчителя, а клас **Student** описує дії учня.

Навчальна програма складається з лекцій і представлена списком **progr.**

Метод ініціалізації у класі **Teacher** завантажує цю програму в атрибут об'єкту **course**.

Після цього всі лекції будуть доступні для об'єкта **novotarskyi** як елементи списку в атрибуті **course**.

Об'єкт **novotarskyi** може використовувати їх для навчання студентів за допомогою методу **teach**.

У цьому методі ми імітуємо процес навчання, заповнюючи атрибути **learn** об'єктів класу **Student**, які присутні на лекції.

Роздрукувавши вкінці вміст атрибутів **learn**, ми можемо отримати інформацію про знання, які кожен із студентів отримав протягом семестру.

```
class Teacher:
    def __init__(self, dis):
          self.course =dis
    def teach(self,num, *pupil):
        for i in pupil:
            if 0<num<=len(self.course):
                i.learn(self.course[num-1])
class Student:
```

```
    def __init__(self):
        self.knowledge = []

    def learn(self, nofl):
        self.knowledge.append(nofl)
progr=['Л1:Типи', 'Л2:Списки', 'Л3:Кортежі',
'Л4:Словники', 'Л5:ООП']
novotarskyi=Teacher(progr)
ivanenko=Student()
petrenko=Student()
sydorenko=Student()
novotarskyi.teach(1, ivanenko, petrenko, sydorenko)
novotarskyi.teach(2, ivanenko)
novotarskyi.teach(3, petrenko,sydorenko)
novotarskyi.teach(12,ivanenko)
print("Іваненко: ",ivanenko.knowledge)
print("Петренко: ",petrenko.knowledge)
print("Сидоренко:",sydorenko.knowledge)
```

7.1.9. Метод **__new__()**

Створення об'єкта відбувається у 2 етапи:

1.**Створення.** Запущено метод **__new__** класу. Цей метод повертає об'єкт цього класу.

2.**Ініціалізація.** Виконується метод класу **__init__**, який ініціалізує вже створений об'єкт. Ми можемо замінити цей метод за замовчуванням, щоб додати код перед ініціалізацією

```
class MyClass:
  def __new__(cls):
    print("Спрацював метод new()")
    return super(MyClass, cls).__new__(cls)
  def __init__(self):
```

```
    print("Спрацював конструктор ")
obj = MyClass()
```

Метод **__new__()** може бути дуже корисним для створення незмінюваних об'єктів або при реалізації патерна Синглтон

(одиночка):

```
class Singleton(object):
  obj = None # единственный экземпляр класса

  def __new__(cls, *args, **kwargs):
    if cls.obj is None:
      cls.obj = object.__new__(cls, *args, **kwargs)
    return cls.obj
single = Singleton()
single.attr = 42
print("single.attr=",single.attr)
newSingle = Singleton()
print("newSingle.attr=",newSingle.attr)
print(newSingle is single) # true
```

Клас є шаблоном об'єкта !!!!

Об'єкт, створений на основі деякого класу, називають **екземпляром класу**.

Види атрибутів

Атрибут класу

Статичний атрибут об’єкту

Динамічний атрибут об’єкту

Методи класу

Системні методи

Прикладні методи

__init__()

__del__()

__new__()

Атрибути об’єкта будемо розділяти на:

- системні атрибути, атрибути користувача.

Системні атрибути – це атрибути, які створюються Пайтоном.

__dict__,

1.Системний атрибут **__dict__** для класу відображає словник атрибутів класу

2.Системний атрибут **__dict__** для об’єкту відображає словник атрибутів об’єкту.

```
class MyClass:pass
print(MyClass.__dict__)
```

Результат:

```
{'__module__': '__main__',…,'__doc__': None}
```

```
class MyClass:
    def __init__(self,at):
        self.a=at
mc=MyClass(10)
print(mc.__dict__)
```

Результат:

```
{'a': 10}
```

Приклад 7.8. Приклад застосування системних атрибутів:

```
class MyClass:
    """Це мій клас"""
    catr=0
    def __init__(self,at):
        self.iatr=at
mc=MyClass(1)
mc.locatr=2
```

```
print("1:", mc.catr,mc.iatr,mc.locatr)  #1
print("2:", mc.__module__)              #2
print("3:", mc.__dict__)                #3
print("4:", mc.__class__)               #4
print("5:", mc.__doc__)                 #5
```

Результат:

```
1: 0 1 2
2: __main__
3: {'iatr': 1, 'locatr': 2}
4: <class '__main__.MyClass'>
5: Це мій клас
```

Атрибути користувача – це атрибути, які створює користувач під час створення класів.

Приклад атрибутів користувача:

Приклад 7.9.

```
class MyClass:
    value = 12
    def __init__(self,f):
        self.one = f
    def mymethod(self,t):
        self.two=t
mc= MyClass(1)
mc.three=3
print("Атрибут класу:",mc.value)
print("Атрибут екземпляра в __init__:",mc.one)
mc.mymethod(2)
print("Починає існування після виклику методу:",mc.two)
print("Атрибут екземпляра динамічний:",mc.three)
```

Результат:

Атрибут класу: 12

Атрибут екземпляра в `__init__`: 1

Починає існування після виклику методу: 2

Атрибут екземпляра динамічний: 3

Атрибут **`__dict__`** *(відображення атрибутів об'єкта)*

Приклад 7.10.

```
class Person:
  Inner= True
  def __init__(self, name, job = None, pay = 0):
    # Конструктор не вимагає трьох аргументів
    self.name = name
    self.job = job
    self.pay = pay
m= Person("Michael","programmer", 100)
print(m.__dict__)
```

Результат:

{'name': 'Michael', 'job': 'programmer', 'pay': 100}

- Атрибут Inner не належить до атрибутів об'єкта **m**

Атрибут **`__dict__`** *(відображення атрибутів класу)*

Приклад 7.11.

```
class Person:
    Inner= True
    def __init__(self,name,job=None, pay = 0):
        self.name = name
        self.job = job
        self.pay = pay
m= Person("Michael","programmer", 100)
print(Person.__dict__)
print(m.__class__.__dict__)
```

Результат:

{'__module__': '__main__', 'Inner': True,..., '__doc__': None}

{'__module__': '__main__', 'Inner': True,..., '__doc__': None}

Створення атрибутів для конкретного екземпляру та доступ до внутрішніх атрибутів класу

Приклад 7.12.

```
class Car:
    wheels=4
    engine=1
my=Car()
my.seats='Leather'
print("Значення атрибуту екземпляру:", my.seats)
print("Значення атрибуту класу:", my.wheels)
print(my.__dict__)
print(my.__class__.__dict__)
```

Результат:

```
Значення атрибуту екземпляру: Leather
Значення атрибуту класу: 4
{'seats': 'Leather'}
{'__module__':'__main__','wheels': 4,'engine':1,... }
```

7.1.10. Поняття про інкапсуляцію

Інкапсуляція - це концепція розробки програмного забезпечення

Ідея інкапсуляції полягає в тому, щоб сховати логіку функціонування від зовнішнього доступу, а користувачеві даного фрагмента коду надати тільки інтерфейс для його використання.

Інкапсуляція важлива при колективному виконанні проєкту групою програмістів.

У цьому випадку проєкт розділяється на частини, які взвємодіють між собою через узгодженя інтерфейси. Такий процес розділення інколи називають нарізкою проєкту.

1. Виникає можливість звертатися до інкапсульованого фрагмента коду з різних областей програми, що заощаджує розмір коду.

2. Якщо необхідно модифікувати інкапсульований фрагмент, то це не вплине на працездатність всієї програми. В Python інкапсуляція виконується за допомогою методів класу.

3. Інкапсуляція в Python працює лише на рівні угоди між програмістами про те, які методи є загальнодоступними, а які - внутрішніми.

4.Одиночне підкреслення на початку імені методу говорить про те, що атрибут або метод не призначений для використання поза методів класу, однак і атрибут і метод з підкресренням доступний за цим іменем.

Приклад 7.13.

```
class MyClass:
    _priatr="Це приватний атрибут"
    def _private(self):
        print("Це приватний метод!")
        print("Не викликайте його ззовні!")
a = MyClass()
print(a._priatr)
a._private()
print(a.__class__.__dict__)
```

Результат:

```
Це приватний атрибут
Це приватний метод!
Не викликайте його ззовні!
{'__module__': '__main__', '_priatr': 'Це приватний
атрибут', '_private': <function MyClass._private at
0x000001454C7EC550>,…}
```

Подвійне підкреслення

Подвійне підкреслення на початку імені атрибута дає більший захист: атрибут стає недоступним за цим іменем

Приклад 7.14.

```
class MyClass:
    def __private(self):
        print("Це реально приватний метод!")
    def exter(self):
        self.__private()
b = MyClass()
b.exter()
#b.__private()
b.MyClass__private()
```

Результат:

```
Це реально приватний метод!
Це реально приватний метод!
```

Приклад розширеного класу Person

Клас `Person`, який не тільки вміє зберігати дані, але й виконувати деякі дії над ними.

Приклад 7.15. Опис розширеного класу

```
class Person:
    def __init__(self, x_name, x_job=None, x_pay=0):
        self.name = x_name
        self.job =  x_job
        self.pay =  x_pay
    def lastName(self): # Метод виводу прізвища
        return self.name.split()[-1]
petro = Person("Petro Petrenko","Python developer", 10000)
ivan = Person("Ivan Ivanov")
```

```
print("Object petro: ",petro.lastName())
print("Object ivan: ", ivan.lastName())
```

Результат: Object petro: Petrenko
Object ivan: Ivanov

В Python є можливість:

1. Перехоплювати за допомогою спеціальних вбудованих методів виконання стандартних операцій
2. Застосовувати нові варіанти цих операцій до екземпляра класу.

Розглянемо виконання перезавантаження операторів на прикладі методу **__str__**.

Цей метод належить типу **class** і викликається, якщо необхідно виконати друк екземпляра класу.

Розглянемо перезавантаження операторів,
модифікувавши клас **Person**.

Приклад 7.16. Перезавантаження методу **__str__**.

```
class Person:
    def __init__(self, x_name, x_job=None, x_pay=0):
        self.name = x_name
        self.job = x_job
        self.pay = x_pay
    def giveRaise(self, percent):
        self.pay = int(self.pay+ (self.pay/100)*percent)
    def __str__(self): # Added method
        return 'Person: %s pay: %s' % (self.name,self.pay)
petro = Person("Petro Petrenko","Python developer", 10000)
ivan = Person("Ivan Ivanov")
petro.giveRaise(10)
print(petro)
print(ivan)
```

Результат виконання:

```
Person: Petro Petrenko pay: 11000
Person: Ivan Ivanov pay: 0
```

7.1.11. Поняття про спадкування

Спадкування є також концепцією розробки. Це один з найголовніших принципів ООП.

Основна мета спадкування: можливість створення ієрархій класів.

Існування ієрархій класів дозволяє істотно скоротити код програмного забезпечення за рахунок запозичення певних властивостей у класів, які є предками даного класу.

Отже, скорочення опису класів за рахунок використання їх подібності відбувається за рахунок створення **ієрархії**.

У корені цієї ієрархії знаходиться базовий клас, від якого походять всі інші класи ієрархії. Ці класи **успадковують** атрибути та методи, уточнюючи й розширюючи поведінку нащадків класу.

Клас «Геометрична фігура» є предком класів «П’ятикутник, шестикутник та семикутник»

Класи «П’ятикутник, шестикутник та семикутник» є нащадками класу «Геометрична фігура»

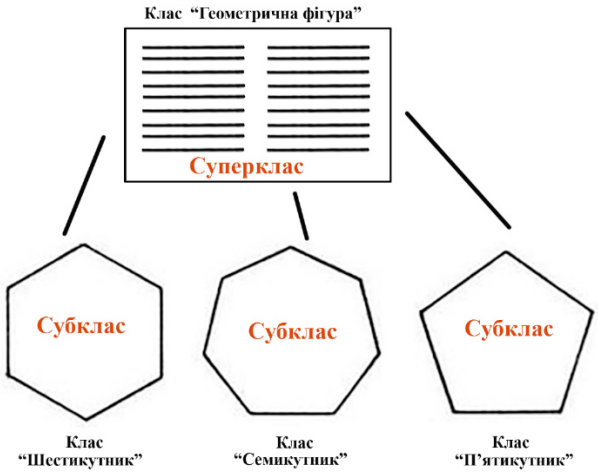


Рис. 7.1. Приклад ієрархії класів

«Геометрична фігура» **суперклас** (надклас)

«П’ятикутник, шестикутник та семикутник» - **субкласи** (підкласи)

Синтаксис спадкування

Клас **OldClass** вказують всередині круглих дужок у визначенні класу **NewClass**:

```
class NewClass(OldClass):
```

Приклад 7.17.

```
class OldClass: pass
class NewClass(OldClass):pass
c = NewClass()
```

Таким чином, клас `NewClass` успадковує всі атрибути й методи класу `OldClass`.

Клас **OldClass** – це базовий або **суперклас**.

Клас **NewClass** – це похідний клас або **підклас.**

Ієрархія спадкування

За допомогою спадкування ми можемо створити новий клас (наприклад, `NewClass`), у якому буде реалізований доступ до всіх атрибутів і методів класу `OldClass`.

Приклад 7.18.

```
class OldClass:
    a_old=1          #Атрибут OldClass
    def m_old(self):#Метод OldClass
        self.a_old=100
        return self.a_old
class NewClass(OldClass):
    a_new=2         #Атрибут NewClass
    def m_new(self):
      self.a_new=3 #Метод NewClass
```

`n=NewClass()`
`print("1:",n.a_old)`
`print("2:",n.a_new)`
	Результат:
	`1: 1`
	`2: 2`

`print("3:",n.m_old())`
`n.m_new()`
`print("4:",n.a_new)`
	`3: 100`
	`4: 3`

Приклад 7.19. Спадкування**.**

```
class Class1: # Базовий клас
    def func1(self):
        print("Метод func1() класу Class1")
    def func2 (self) :
        print("Метод func2() класу Class1")
class Class2(Class1):# Клас Class2 успадковує клас Class1
    def func3 (self) :
        print("Метод func3() класу Class2")
c = Class2 ()   # Створюємо екземпляр класу  Class2
c.func1()       # Виведе: Метод func1() класу  Class1
c.func2()       # Виведе: Метод func2() класу Class1
c.func3()       # Виведе: Метод func3 () класу Class2
```

Результат виконання:

```
Метод func1() класу Class1
Метод func2() класу Class1
Метод func3() класу Class2
```

Механізм перевизначення методів

Якщо ім'я методу в класі **NewClass** збігається з ім’ям методу класу **OldClass**, то буде використовуватися метод з класу **NewClass**. – це є механізм перевизначення методів.

Щоб викликати однойменний метод з базового класу необхідно:

1. Указати перед методом назву базового класу.
2. У першому параметрі методу необхідно явно вказати `self`.

Якщо маємо спадкування:

```
class NewClass(OldClass):
```

Якщо в обидвох класах існує метод:

```
def mymet(self):
```

тоді з класу **NewClass** можна викликати метод **mymet** так:

```
OldClass.mymet(self)
```

Приклад 7.20. Перевизначення методів

```
class OldClass:       # Базовий клас
    def __init__(self):
        print("Конструктор базового класу")
    def func(self):
        print("Метод func() класу OldClass")
class NewClass(OldClass): # Клас NewClass успадковує клас OldClass
    def __init__(self):
       print("Конструктор підкласу")
       OldClass.__init__(self) # Викликаємо конструктор
базового класу
    def func(self):
        print("Метод func() класу NewClass")
        OldClass.func(self)# Викликаємо метод базового класу
c = NewClass()     # Створюємо екземпляр класу NewClass
c.func()   # Викликаємо метод func ()
```

Результат виконання:

```
Конструктор підкласу
Конструктор базового класу
Метод func() класу NewClass
Метод func() класу OldClass
```

УВАГА!

Якщо у підкласі є конструктор то конструктор базового класу не викликається оскільки він перевизначений у підкласі.

```
# Викликаємо конструктор базового класу
OldClass.__init__(self)
```

Щоб викликати однойменний метод з базового класу, можна також скористатися функцією **super().**

Формат функції:

super([<**Поточний класс**>, <**self**>])

Два способи застосування

1.super().__init__() # Викликаємо конструктор базового класу

2.super(**NewClass, self**).**__init__()**

#Викликаємо конструктор базового класу

При використанні функції `super()` не потрібно явно передавати вказівник `self` у метод, який викликаємо.

super().__init__()

Крім того, у першому параметрі функції `super()` вказують поточний клас (підклас), а не базовий.

super(**NewClass,** self).__init__()

Пошук ідентифікатора буде проводитися у всіх базових класах. Результатом пошуку стане перший знайдений ідентифікатор у ланцюжку спадкування.

Приклад 7.21.

```
class OldClass:      # Базовий клас
    def func(self):
        print("Метод func() класу OldClass")
class MidClass(OldClass):
    def func(self):
        print("Метод func() класу MidClass")
class NewClass(MidClass):
    def func(self):
       print("Метод func() класу NewClass")
       super().func()
c = NewClass()
c.func()  # Викликаємо метод func ()
```

Механізм перевизначення атрибутів

Якщо ім'я атрибуту в класі **NewClass** збігається з ім'ям атрибуту класу **OldClass**, то буде використовуватися атрибут з класу **NewClass** – це і є механізм перевизначення атрибутів.

Для доступу до атрибуту базового класу у підкласі в такому випадку необхідно явно вказувати ім'я базового класу:

Приклад 7.22.

```
class OldClass:
    x,y=10,100
class NewClass(OldClass):
    x,y=20,200
    z=OldClass.y
c = NewClass()
print('x=',c.x)
print('y=',c.y)
print('z=',c.z)
```

Результат: x= 20
y= 200
z= 100

Перевизначення стандартного класу

При перевизначенні стандартного класу наш підклас клас містить всі його атрибути, при цьому деякі з них можуть бути перевизначені або додані. Наприклад, створюємо свій клас, схожий на словник.

Приклад 7.23.

```
class Mydict(dict):
    def get(self, key, default = 0):
        return dict.get(self, key, default)
a = dict(a=1, b=2)
b = Mydict(a=1, b=2)
b['c'] = 4
```

```
print(b)
print(a.get('v'))
print(b.get('v'))
```

Результат:

```
{'a': 1, 'b': 2, 'c': 4}
None
0
```

Множинне спадкування

У визначенні класу в круглих дужках можна вказати відразу кілька базових класів через кому. Розглянемо множинне спадкування на прикладі.

Приклад 7.24.

```
class Class1: # Базовий клас для класу Class2
    def func0 ( self) :
        print("Метод func0() класу Class1")
    def func1 ( self) :
        print("Метод func1() класу Class1")
class Class2(Class1): # Клас Class2 успадковує клас Class1
    def func2(self):
        print("Метод func2() класу Class2")

class Class3(Class1): # Клас Class3 успадковує клас Class1
    def func1(self):
        print( "Метод func1 () класу Class3")
    def func2(self):
        print("Метод func2() класу Class3")
    def func3(self):
        print("Метод func3() класу Class3")
    def func4(self):
        print("Метод func4() класу Class3")
class Class4(Class2, Class3):# Множинне спадкування
```

```
    def func4 (self):
        print("Метод func4() класу Class4")
c = Class4() # Створюємо екземпляр класу Class4
c. func0()   # Виведе: Метод func0 () класу Class1
c. func1()   # Виведе: Метод func1 () класу Class3
c. func2()   # Виведе: Метод func2 () класу Class2
c. func3()   # Виведе: Метод func3 () класу Class3
c. func4()   # Виведе: Метод func4 () класу Class4
```

Метод **func0()** визначений у класі: **Class1.**

Метод **func1()** визначений у двох класах: **Class1** і **Class3.**

Оскільки спочатку відбувається перегляд всіх базових класів, безпосередньо зазначених у визначенні поточного класу, метод `func1()` буде знайдений у класі `Class3`, оскільки він зазначений у числі базових класів у визначенні `Class4`.

Метод **func2()** також визначений у двох класах: `Class2 i Class3.` Оскільки клас `Class2` розміщений першим у списку базових класів, то метод буде знайдений саме в ньому. Щоб одержати доступ до методу із класу `Class3`, слід указати це явно.

Змінимо визначення класу `Class4` з попереднього прикладу й успадковуємо метод `func2()` з класу `Class3.`

Приклад 7.25.

```
class Class4(Class2, Class3):# Множинне спадкування
# Успадковуємо func2() із класу Class3, а не із класу Class2
    func2 = Class3.func2
    def func4(self):
        print("Метод func4() класу Class4")
        c = Class4()
        c. func2()
```

Метод **func3()** визначений тільки в класі Class3, тому метод успадковується від цього класу. Метод func4(), визначений у класі Class3, перевизначається в підкласі Class4.

Якщо ж шуканий метод знайдений у підкласі, то вся ієрархія спадкування переглядатися не буде.

Для одержання переліку базових класів можна скористатися атрибутом __bases__.

Як значення атрибут повертає кортеж. Як приклад виведемо базові класи для всіх класів з попереднього прикладу:

Приклад 7.26.

```
print(Class1.__bases__)
print(Class2.__bases__)
print(Class3.__bases__)
print(Class4.__bases__)
```

Результат виконання:

```
(<class 'object'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class2'>, <class '__main__.Class3'>)
```

Розглянемо порядок пошуку ідентифікаторів при складній ієрархії множинного спадкування

Приклад 7.27.

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class1): pass
class Class7(Class4, Class6): pass
```

```
c = Class7 ()
print(c.x)
Результат виконання: 10
```

Послідовність пошуку атрибутів буде такою:

Class7->Class4->Class3->Class2->Class6->Class1

Одержати весь ланцюжок спадкування дозволяє атрибут **__mro__**:

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class1): pass
class Class7(Class4, Class6): pass
c = Class7 ()
print(Class7. __mro__)
```

Результат виконання:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>,
<class '__main__.Class3'>, <class '__main__.Class2'>,
<class '__main__.Class6'>, <class '__main__.Class1'>,
<class 'object'>)
```

Робота атрибута **__mro__**

Атрибут __mro__ призначений для знаходження найкоротшого шляху й ланцюжка спадкування.

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class1): pass
class Class7(Class4, Class6): pass
```

```
c = Class7 ()
print(Class7. __mro__)
(<class '__main__.Class7'>, <class '__main__.Class4'>,
<class '__main__.Class3'>, <class '__main__.Class2'>,
<class '__main__.Class6'>, <class '__main__.Class1'>,
<class 'object'>)
```

Розглянемо ланцюжки спадкування для класу **Class7:**

Class7–Class4–Clas3-Class2-Class1-x

Class7–Class6–Class1-x

7.1.12. Поліморфізм

Парадигма об'єктно-орієнтованого програмування крім **інкапсуляції та спадкування** включає ще одну важливу особливість — поліморфізм.

Слово «поліморфізм» можна перекласти як «багато форм».

В ООП (об’єктно-орієнтованому програмуванні) цим терміном позначають:

можливість використання того самого імені операції або методу до об'єктів різного типу,

при цьому дії, виконувані з об'єктами, можуть суттєво різнитися в залежності від їх поточного типу. Тому можна сказати, що в одного імені багато форм.

Поліморфізм за рахунок поліморфності операцій в мові Python

Приклад 7.28. Приклад поліморфної операції «+»

```
class First:
    def func(self,a,b):
        return a+b
m=First()
print(m.func(25,75))
print(m.func("При", "віт"))
print(m.func(True, True))
```

```
print(m.func((1,2), (3,4)))
print(m.func([5,6], [7,8]))
Результат: 100
           Привіт
           2
          (1, 2, 3, 4)
          [5, 6, 7, 8]
```

Використовуємо один і той же метод, який дає різну реакцію при вводі різних даних

Поліморфізм класів зі статичною типізацією

Поліморфізм класів виникає у випадку, коли маємо однойменні методи у різних класах.

Наприклад. Два різні класи можуть містити метод з назвою **total.** Інструкції у цих методах можуть передбачати зовсім різні операції:

1. Клас `FClass` містить метод `total`, який виконує додавання до вхідного параметру числа 10.
2. Клас `SClass` містить метод **total**, який виконує підрахунок кількості символів у вхідному параметрі.

Залежно від того, до об'єкта якого класу застосовується метод **total**, виконуються ті або інші інструкції.

Приклад 7.29. Маємо два об’єкти різного типу

```
class FClass:
     n=10
     def total(self,N):
          self.total = int(self.n) + int(N)

class SClass:
     def total(self,s):
          self.total = len(str(s))
```

```
f = FClass()
s = SClass()
f.total(45)
s.total(45)
print (f.total) # Вивід: 55
print (s.total) # Вивід: 2
Результат виконання: 55 2
```

Поліморфізм класів з динамічною типізацією

Програма вимагає введення числа користувачем. Якщо число належить до діапазону від -100 до 100, то створюється об'єкт одного класу `One`.

Якщо число не належить до діапазону від -100 до 100, то створюється об'єкт класу `Two`.

В обох класах задамо метод-конструктор **`__init__`**, який у першому класі підносить число до квадрату, у другому класі множить на два введене число.

Приклад 7.30. Один об’єкт з динамічною зміною типу

```
class One:
    def __init__(self,a): self.a = a ** 2
    def __str__(self):
        return '[Піднесли до квадрата: %s]' % (self.a)
class Two:
    def __init__(self,a): self.a = a * 2
    def __str__(self):
        return '[Подвоїли: %s]' % (self.a)
a = input ("Введіть число: ")
a = int(a)
if -100 < a < 100: obj = One(a)
else:
    obj = Two(a)
```

```
print (obj)


obj1 = One(45)
obj1.out()
obj1.multi(2)
```

Результат:

```
Введіть число: 12
[До квадрата: 144]
Введіть число:  128
[Подвоїли: 256]
```

7.1.13. Перевизначення методів

Використання поліморфізму та спадкування класів дозволяє перевизначати методи суперкласів у підкласах. **Наприклад,** може виникнути ситуація, коли один підклас використовує визначений метод із суперкласу, а другий підклас перевизначає цей метод. У цьому випадку метод перевизначають в підкласі.

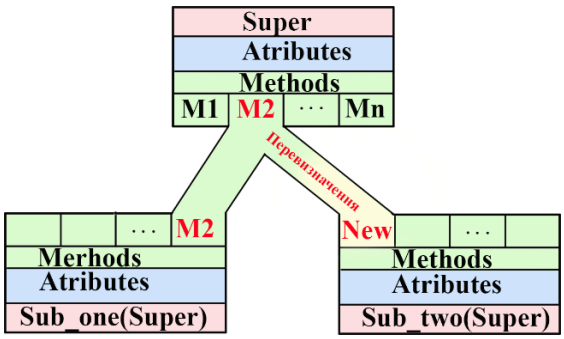


Рис. 7.2. Перевизначення методів

Приклад 7.31.

```
class Base:
    def __init__(self,n):
        self.numb = n
    def out(self):
        print (self.numb)
```

```
class One(Base):
     def multi(self,m):
          self.numb *= m
class Two(Base):
     def inlist(self):
          self.inlist = list(str(self.numb))
     def out(self):
          for i in self.inlist:
             print(i)
```

Випадок1.
```
obj1 = One(45)
obj1.out()
```

45

90

4

5

Об'єкт **obj1** використовує методи `__init__()` і **out()** з суперкласу **Base**,

Випадок2.
```
obj1.multi(2)
obj1.out()
```

Об'єкт **obj1** використовує методи `multi()` з класу `One` і метод **out()** з суперкласу **Base**,

Випадок3.
```
obj2 = Two(45)
obj2.inlist()
obj2.out()
```

Об'єкт **obj2** використовує метод `__init__()` з суперкласу **Base**, а методи `inlist()` і **out()** з класу `Two`

Атрибути шукають «знизу нагору»: спочатку в класах, потім суперкласах. Оскільки для **obj2** атрибут **out** уже був знайдений у класі **Two**, то з класу **Base** він не використовується. Інакше кажучи, клас **Two** перевизначає атрибут суперкласу **Base**.

7.1.14. Домішки і їх використання

Множинне спадкування в Python, дозволяє реалізувати цікавий спосіб розширення функціональності класів за допомогою так званих ***домішок*** (**`mixings`**). Домішка – це клас, що включає які-небудь атрибути й методи, які необхідно додати до інших класів.

Оголошуються вони подібно до звичайних класів.

Приклад 7.32 Клас-домішка

```
class Mixing:     # Визначаємо сам клас-домішку
  attr = 0            # Визначаємо атрибут домішки
  # Визначаємо метод домішки
  def mixing_method(self):
    print("Метод домішки")
m = Mixing  #Зверніть увагу. Дужок немає
m.mixing_method(m)
```

```
Результат виконання:  Метод домішка
```

Це спеціальний клас, який використовуємо тільки для розширення функціональності інших класів. Але для цього класу також можливо утворювати об’єкти.

Тепер оголосимо два класи, додамо до їхньої функціональності ту, що визначена в класі домішці **`Mixing`**, і перевіримо її в дії.

Приклад 7.33. Розширення функціональності класів за допомогою домішки

```
class MixClass(Mixing):
  def MC_method(self):
    print("Метод класу MixClass")
class SubClass (MixClass):
  def Sub_method(self):
    print("Метод класу SubClass")
c1=MixClass()
```

```
c1.MC_method()
c1.mixing_method ()
c2=SubClass()
c2.MC_method()
c2.Sub_method()
c2.mixing_method()
```

```
Результат виконання:
Метод класу MixClass
Метод домішки
Метод класу MixClass
Метод класу SubClass
Метод домішки
```

Сфера застосування домішок

1.Домішки активно застосовуються в різних додаткових бібліотеках – зокрема, у популярній бібліотеці Web-програмування Django.

2. Застосування домішок в графічних та текстових редакторах дозволяє зробити кнопки керування багато функціональними.

Нижче розглянемо приклад

1. В залежності від значення параметра об’єкт стає екземпляром іншого класу.
2. У методі класу `Subclass` викликають метод `out()`
3. У методі `out()` викликають метод `out()` суперкласу

Приклад 7.34

```
class Base:
    def __init__(self,N):
          self.numb = N
    def out(self):
        self.numb /= 2
        if type(self) is Base:
            print("Base:", self.numb, end=" ")
```

```
        elif type(self) is Subclass:
            print("\nSubclass:",self.numb)
class Subclass(Base):
     def out(self):
          self.numb*=2
          Base.out(self)
for i in range(15):
    if 4 < i < 10:
          obj = Subclass(i)
    else:
          obj = Base(i)
    obj.out()
```

Рузультат роботи:

```
Base: 0.0 Base: 0.5 Base: 1.0 Base: 1.5 Base: 2.0
Subclass: 5.0
   Subclass: 6.0
   Subclass: 7.0
   Subclass: 8.0
   Subclass: 9.0
   Base: 5.0 Base: 5.5 Base: 6.0 Base: 6.5 Base: 7.0
```

7.2. Стандартні функції і методи об’єктів

7.2.1. Функція getattr()

Функція `getattr()` – повертає значення атрибута по його назві, заданій у **вигляді рядка**.

За допомогою цієї функції можна сформувати ім'я атрибута динамічно під час виконання програми. Формат функції:

getattr(<Об’єкт>, <Атрибут>[, <Знач. за замовч>])

Якщо зазначений атрибут не знайдений, виконується виключення **`AttributeError`.**

Щоб уникнути виводу повідомлення про помилку, можна в третьому параметрі вказати значення, яке буде повертатися, якщо атрибут не існує.

Застосування функції `getattr()`

Приклад 7.35.

```
class Base:
    first = 10
class One(Base):
    second = 20
class Two(Base):
    third = 30
x = Base()
y = One()
z = Two()
L = [(x, "first"), (y, "second"), (z, "third"), (z,
"fourth")]
for i,j in L:
  print("{0:6} = {1:d}".format(j,getattr(i, j, 0)))
```

Результат роботи :

```
first  = 10
second = 20
third  = 30
fourth = 0
```

Приклад 7.36.

```
class Base:
    aaa = 10
class One(Base):
    aa = 20
class Two(Base):
    a = 30
atr=""
```

```
mytuple=(Two,One,Base)

for Cl in mytuple:
  myobj=Cl()
  atr+='a'
print("{0}.{1}={2}".format(myobj.__class__.__name__,atr,getattr(myobj,atr)))
```

Результат:

Two.a = 30

One.aa = 20

Base.aaa = 10

7.2.2. Функція **setattr()**

Функція `setattr ()` – задає значення атрибута. Назва атрибута вказується **у вигляді рядка**.

Формат функції:

setattr(<**Об'єкт**>, <**Атрибут**>, <**Значення**>)

Другим параметром методу `setattr()` можна передати ім'я неіснуючого атрибута. Якщо атрибут не існує, то він буде створений із зазначеним іменем.

Приклад 7.37. Застосування функції **setattr()**

```
class One:
     to=100
class Two:
     tw=300
x=One()
y=Two()
L=[(x,"to",1), (x,"td",2),(y,"tw",3),(y,"td",4)]
for i,j,k in L:
    setattr(i,j,k)
print(x.__dict__)
print(y.__dict__)
```

Результат роботи:

```
{'to': 1, 'td': 2}
{'tw': 3, 'td': 4}
```

Приклад 7.38.

1. Задамо три класи, які пов’язані спадкуванням.
2. Створимо об’єкт `obj` типу `Two`
3. Створимо два списки

```
class Zero:
     atr0= 0
class One(Zero):
     atr1= 100
class Two(One):
    atr2 = 200
obj=Two()
mylist=[Zero,One,Two]
mydata=[1,2,3,4]

for i in range(4):
   catr = "atr" + str(i)
   if i<len(mylist):
      setattr(mylist[i], catr, mydata[i])
      print(mylist[i].__name__+"."+catr,"=",getattr(obj,catr))
   else:
      setattr(mylist[2], catr, mydata[i])
      print(mylist[2].__name__+"."+catr,"=",getattr(obj,catr))
```

Результат роботи:

```
Zero.atr0 = 1
One.atr1 = 2
Two.atr2 = 3
Two.atr3 = 4
```

7.2.3. Функція **delattr()**

Функція delattr (<Об'єкт>, <Атрибут>) – видаляє зазначений атрибут. Назва атрибута вказується **у вигляді рядка**.

Приклад 7.39

```
class One():
  def __init__(self,a,b):
         self.t3 = a
         self.t4 = b
x=One(20,50)
print("Атрибути до видалення",x.__dict__)
delattr(x,"t3")
print("Атрибути після видалення",x.__dict__)
```

Результат роботи:

```
Атрибути до видалення:{'t4': 50, 't3': 20}
Атрибути після видалення: {'t4': 50}
```

Функція hasattr (<Об'єкт>, <Атрибут>) – перевіряє наявність зазначеного атрибута. Якщо атрибут існує, функція повертає значення True.

Назва атрибута вказується **у вигляді рядка**. Продемонструємо роботу функцій на прикладі.

Приклад 7.40

```
class MyClass:
   def __init__(self):
        self.x = 10
   def set_y(self,a):
        self.y=a
        return self.y
c = MyClass ()
print(hasattr(c, "x"))   # Виведе: True
print(hasattr(c, "y"))   # Виведе: False
```

```
c.set_y(5)
print(hasattr(c, "y"))    # Виведе: True
```

7.2.4. Принципи доступу до атрибутів

1.Усі атрибути класу в мові Python є відкритими (**public**)

Отже атрибути є доступними для безпосередньої зміни:

- із самого класу,
- з інших класів,
- з основного коду програми.

Атрибути можна створювати динамічно. Можна створити як атрибут класу, так і атрибут екземпляра класу. Розглянемо це на прикладі:

Приклад 7.41. Атрибути класу й екземпляра класу

```
class MyClass:  # Визначаємо порожній клас
    pass
MyClass.x = 50 # Створюємо атрибут об’єкта класу
# Створюємо два екземпляри класу
c1, c2 = MyClass(), MyClass()
c1.__class__.z = 100
c1.y = 10  # Створюємо атрибут екземпляра класу
c2.y = 20  # Створюємо атрибут екземпляра класу
print(c1.x, c1.y)  # Виведе: 50 10
print(c2.x, c2.y)  # Виведе: 50 20
print(c1.z)# Виведе: 100
print(c2.z)# Виведе: 100. Отже z-атрибут класу
```

У цьому прикладі ми визначаємо порожній клас, розмістивши в ньому оператор **pass**.

Далі створюємо атрибут класу **x**. Цей атрибут буде доступний усім створюваним екземплярам класу. Потім створюємо два екземпляри класу і додаємо однойменні атрибути **y**.

Значення цих атрибутів будуть різними в кожному екземплярі класу.

Але якщо створити новий екземпляр (наприклад, `c3`), то атрибут `y` в ньому визначений не буде. Таким чином, за допомогою класів можна імітувати типи даних, підтримувані іншими мовами програмування (наприклад, тип `struct`, доступний у мові C).

Дуже важливо розуміти різницю між атрибутами класу й атрибутами екземпляра класу.

7.2.5. Визначення атрибутів класу та атрибутів екземпляра класу

Атрибут класу доступний усім екземплярам класу, але після зміни атрибута значення зміниться у всіх екземплярах класу.

Атрибут екземпляра класу може зберігати унікальне значення для кожного екземпляра, і зміна його в одному екземплярі класу не торкнеться значень однойменного атрибута в інших екземплярах того ж класу. Розглянемо це на прикладі, створивши клас з атрибутом класу (`x`) і атрибутом екземпляра класу (`y`):

Приклад 7.42. Зміна атрибута класу

```
class MyClass:
    x1 = 10  # Атрибут класу
    x2 = 20
c1 = MyClass() # Створюємо екземпляр класу
c2 = MyClass() # Створюємо екземпляр класу
print(c1.x1, c2.x1)# 10 10
print(c1.x2, c2.x2)# 20 20
c1.__class__.x1=88 # Змінюємо атрибут класу
MyClass.x2=98      # Через ім'я класу
print(c1.x1, c2.x1) # 88 88
print(c1.x2, c2.x2) # 98 98
```

Як видно з прикладу, зміна атрибута класу торкнулася значення у двох екземплярах класу відразу.

Зміна атрибута екземпляра класу

Приклад 7.43.

```
class MyClass:
   def __init__(self,a):
        self.y = a # Атрибут екземпляра класу
c1 = MyClass(20) # Створюємо екземпляр класу
c2 = MyClass(40) # Створюємо екземпляр класу
print(c1.y, c2.y)  # 20 40
c1.y = 88 # Змінюємо атрибут екземпляра класу
print(c1.y, c2.y)# 88 40 Зміна в екземплярі c1.
```

Результат:

20 40

88 40

Атрибут екземпляра може бути заданий безпосередньо в екземплярі класу.

Приклад 7.44.

```
class MyClass:
    x = 10  # Атрибут класу
    def __init__(self,a):
         self.y=a
c1 = MyClass(1)  # Створюємо екземпляр класу
c2 = MyClass(2)  # Створюємо екземпляр класу
print(c1.y, c2.y)
print(getattr(c1,"y",100),getattr(c2,"z",200))
setattr(c1,"y",100)
setattr(c1,"z",10)
setattr(c2,"z",300)  # Змінюємо атрибут класу
print(getattr(c1,"y"),getattr(c2,"z"))
print("c1.y:",c1.y,"c2.y:",c2.y,"c1.z:",
c1.z,"c2.z:",c2.z)
```

1 2

1 200

100 300

c1.y: 100 c2.y: 2 c1.z: 10 c2.z: 300

Одночасне існування атрибута класу та екземпляра класу з однаковим іменем

При одночасному існуванні атрибутів класу та екземпляра класу с одним іменем доступ до них потрібно виконувати, користуючись таким правилом.

1. Доступ до атрибута класу виконувати з безпосереднім вказуванням імені класу через крапку.
2. Доступ до атрибута екземпляра класу виконувати з безпосереднім вказуванням імені екземпляра класу через крапку.

Приклад використання атрибутів класу та екземпляра класу з однаковим іменем

Приклад 7.45.

```
class MyClass:
    x = 10  # Атрибут класу
c = MyClass() # Створюємо екземпляр класу
c.__class__.x=100
# MyClass.x = 100    # Змінюємо атрибут об'єкта класу
print(c.x) # 100 # Вивели атрибут класу
c.x = 200  # Створили атрибут екземпляра класу!!
print(c.x, c.__class__.x)
# print(c.x, MyClass.x)  # 200 100
```

7.2.6. Спеціальні методи

Метод **__call__()**

Метод `__call__()` викликається при виклику екземпляра класу, як виклик функції.

Формат методу:

def **__call__**(self[, <Парам1>[,...,<Парамn>]])

Приклад 7.46. Мінімальний код

```
class A:
   def __call__(self):
       print("Виклик у стилі функції")
a=A()
a()
```

Результат:

Виклик у стилі функції

Приклад 7.47. Застосування методу **__call__**

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __call__(self,n):
        print(self.msg+n)
        return "Повертаємо "+self.msg
x = MyClass("Повідомлення об'єкта x")
# Створення екземпляра класу
y = MyClass("Повідомлення об'єкта y")
# Створення екземпляра класу
x("!!")
b=y("!")
print(b)
```

Результат: Повідомлення об'єкта x !!

Повідомлення об'єкта y !

Повертаємо повідомлення y

Метод **__getattr__()**

Метод __getattr__() – викликається при спробі доступу до неіснуючого атрибута класу. Формат методу:

```
def __getattr__(self, <Атрибут>)
```

Приклад 7.48. Мінімальний код

```
class A:
    def __getattr__(self,b):
      print("Відсутній атрибут:",b)
      return "Ви помилились"
a=A()
print(a.c)
print(a.myatr)
```

Результат:

Відсутній атрибут: c

```
Ви помилились
Відсутній атрибут: myatr
Ви помилились
```

Приклад 7.49.

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print("Спрацював метод __getattr__()")
        return "Щось виводимо через return"
c = MyClass()
# Атрибут і існує
print(c.i)
print(c.s)
```

Результат:

```
20
Спрацював метод __getattr__()
Щось виводимо через return
```

Метод **__getattribute__()**

(«декоратор зчитування атрибута»)

Метод викликається при доступі до будь-якого атрибута класу.

Формат методу:

```
def __getattribute__(self, <Атрибут>)
```

Необхідно враховувати, що використання точкової нотації (для доступу до атрибута класу) всередині цього методу призведе до зациклення. Щоб уникнути зациклення, слід викликати метод `__getattribute__()` об'єкта `object`. Всередині методу потрібно повернути значення атрибута або виконати виключення `AttributeError`.

Приклад 7.50.

```
class MyClass:
    def __init__(self):
```

```
        self.i = 20
        self.k = 30
    def __getattribute__(self, a):
        print("Викликано __getattribute__()")
        return object.__getattribute__(self, a)   # Тільки так!!!
c = MyClass ()
print(c.i)
print(c.k)
```

Результат:

```
Викликано __getattribute__()
20
Викликано __getattribute__()
30
```

Метод **__setattr__()**

(«декоратор створення атрибута»)

Метод викликається при спробі присвоювання значення атрибуту екземпляра класу.

Формат методу:

def __setattr__(self,<**Атрибут**>,<**Значення**>)

Якщо всередині методу необхідно присвоїти значення атрибуту, то слід використовувати словник __dict__, інакше при точковій нотації метод __setattr__() буде викликаний повторно, що призведе до зациклення.

Приклад 7.51.

```
class MyClass:
    def __setattr__(self, a, value):
        print("Метод __setattr__()")
        if value%2==0:
            self.__dict__[a] = value+128#Тільки через словник !!!
        else:
```

```
            self.__dict__[a] = value + 256  # Тільки через
```
словник !!!

```
c = MyClass()
c.i = 10
print(c.i)
c.i = 11
print(c.i)
```

Результат:

```
Метод __setattr__()
138
Метод __setattr__()
267
```

Метод **`__len__()`**

Метод викликають при використанні функції `len()`

Формат методу :

```
def  __len__ (self)
```

Метод повинен повертати додатне ціле число.

Приклад 7.51.

```
class MyClass:
    x = "Line"
    def __len__(self):
        b = len(self.x)
        print("Метод __len__()")
        return b
c = MyClass()
print(len(c))
```

Результат:

```
Метод __len__()
4
```

Метод `__bool__(self)` – викликається при використанні функції `bool()`.

```
class MyClass:
     def __bool__(self):
        print("Викликано bool")
        return True
c = MyClass()
a=bool(c)
```

Метод `__int__ (self)` – викликається при перетворенні об'єкта в ціле число за допомогою функції `int()`.

```
class MyClass:
     def __int__(self):
        print("Викликано int")
        return 0
c = MyClass()
a=int(c)
```

Метод `__float__(self)` – викликається при перетворенні об'єкта в дійсне число за допомогою функції `float()`.

```
class MyClass:
     def __float__(self):
        print("Викликано float")
        return 0.1
c = MyClass()
a=float(c)
```

Метод `__complex__(self)` – викликають при перетворенні об'єкта в комплексне число за допомогою функції `complex()`.

```
class MyClass:
     def __complex__(self):
        print("Викликано complex")
        return 1+1j
```

```
c = MyClass()
a=complex(c)
```

Метод `__round__(self, n)` – викликають при використанні функції `round()`.

```
class MyClass:
    def __round__(self, n):
        print("Викликано round")
        return n
c = MyClass()
a = round(c, 2)
print(a)
```

Результат: Викликано round

2

Метод `__index__(self)` – викликають при використанні функцій `bin()`, `hex()` і `oct()`.

```
class MyClass:
     def __index__(self):
        print("Викликано index")
        return 0b0101
c = MyClass()
a=bin(c)
```

Метод `__repr__()` викликають при виводі в інтерактивній оболонці, а також при використанні функції `repr()`.

```
class MyClass:
     def __repr__(self):
        print("Викликано repr")
        return "a"
c = MyClass()
a=repr(c)
```

Метод `__str__()` викликають при виводі за допомогою функції `print()`, а також при використанні функції `str()`.

```
class MyClass:
     def __str__(self):
         print("Викликано str")
         return "a"
c = MyClass()
a=str(c)
```

Якщо метод `__str__()` відсутній, то буде викликаний метод `__repr__()`.

Як значення методи `__repr__()` і `__str()__` повинні повертати рядок.

Приклад 7.52.

```
class MyClass:
  def __init__(self, m):
    self.msg = m
  def __repr__(self):
    return "Метод __repr__() {0}".format(self.msg)
  def __str__ (self):
    return "Метод __str__() {0}".format(self.msg)
c = MyClass("Значення")
print(repr(c))
print(str(c))
print(c)
```

Результат роботи:

```
Викликаний метод __repr__() Значення
Викликаний метод __str__() Значення
Викликаний метод __str__() Значення
```

Метод **`__hash__(self)`**

Метод потрібно перевизначити, якщо екземпляр класу заплановано використовувати як ключ словника або всередині множини.

Приклад 7.53.

```
class MyClass:
    def __init__ (self, y):
        self.x = y
```

```
    def __hash__ (self) :
        return hash(self.x)
m = MyClass(10)
d = {}
d[m] = "Значення"
print(d[m])
print(d)
Результат: Значення
{<__main__.MyClass object at 0x01CF0B90>: 'Значення'}
```

Контрольні запитання до розділу 7.

1. Перерахуйте відомі вам парадигми програмування.
2. Які існують концепції розробки ООП?
3. Дайте визначення класу.
4. Поясніть необхідність використання методу __int__().
5. Дайте приклад застосування методу __new__()
6. Чим відрізняються принципи використання системних атрибутів та атрибутів користувача?
7. Поясніть властивості інкапсуляції.
8. Поясніть поняття та синтаксис спадкування.
9. Що означає множинне спадкування?
10. Опишіть механізм перевизначення методів.
11. Опишіть умови, коли використовують домішки.
12. Стандартні функції і методи об’єктів.
13. Перерахуйте відомі вам спеціальні методи?
14. Дайте приклад, використання методу __str__().
15. Чим відрізняється метод __str__() і метод __repr__().

8. РОБОТА З ФАЙЛАМИ

8.1. Відкриття файлу

Перш ніж працювати з файлом, необхідно створити об'єкт файлу **f** за допомогою функції **open().** Функція має наступний формат:

f = open(<**Шлях до файлу**>[, mode='r']
[, buffering=-1] [, encoding=None][,errors =None] [, newline=None] [, closefd=True]**)**

У першому параметрі вказують шлях до файлу. Шлях може бути абсолютним або відносним. Задаючи абсолютний шлях до файлу в Windows слід враховувати, що в **Python** зворотний слеш є спеціальним символом. Тому зворотний слеш необхідно подвоювати або замість звичайних рядків використовувати неформатовані рядки.

8.1.1. Абсолютний шлях до файлу

Абсолютний шлях до файлу – це шлях, який починається з імені диску і закінчується іменем файлу.

Приклад 8.1.

```
C:\\temp\\new\\file.txt" # Правильно
r"C:\temp\new\file.txt" # Правильно
"C:\temp\new\file.txt"  # Неправильно!!
print(repr("C:\temp\new\file.txt"))
```

Результат: `'C:\temp\new\x0cile.txt'`

У цьому шляху через те, що слеші не подвоєні, виникла присутність відразу трьох спеціальних символів: `\t`, `\n` і `\f` (відображається як `\x0c`). Після перетворення цих спеціальних символів шлях буде мати вигляд:

C:<Табуляція>emp<Переведення рядка>ew <Переведення формату>ile.txt

Виключення при доступі до файлу

Якщо в якості параметра функції **open()** передати помилковий рядок, то це призведе до виключення **OSError**:

```
f=open("C:\temp\new\file.txt")
```

```
Traceback (most recent call last):
  File "C:/PYTHON/Lecture25/Ex1.py", line 7, in <module>
    f=open("C:\temp\new\file.txt")
OSError: [Errno 22] Invalid argument:
'C:\temp\new\x0cile.txt'
```

8.1.2. Відносний шлях до файлу

Замість абсолютного шляху до файлу можна вказати відносний шлях. У цьому випадку шлях визначають з врахуванням розташування поточного робочого каталогу.

Поточним робочим каталогом будемо називати папку проєкту.

Наприклад, якщо папка проєкту розташована на шляху "C:\PYTHON",
то при задаванні відносного шляху

```
r"folder1\folder2\file.txt"
```

повний шлях до файлу матиме вигляд:

```
r"C:\PYTHON\ folder1\folder2\file.txt"
```

8.1.3. Функція **abspath()**

Відносний шлях можна перетворити в абсолютний шлях за допомогою функції **abspath()** з модуля: **os.**

Можливі наступні варіанти:

1. Файл в поточному робочому каталозі. Можна вказати тільки назву файлу.

Приклад 8.2.

```
import os.path as pt
a=pt.abspath(r"potoch.txt")
print(a)
```

Результат:

```
C:\PYTHON\potoch.txt
```

2.Файл у вкладеній папці. Перед назвою файлу потрібно вказати назви вкладених папок, розділяючи їх зворотним слешем.

Приклад 8.3.

```
import os.path
pt=os.path.abspath(r"folder1\f1.txt")
```

```
print(pt)
pp=os.path.abspath(r"folder1\folder2\f2.txt")
print(pp)
pm=os.path.abspath(r"vkladena\vkladfil.txt")
print(pm)
```

Результат:

```
C:\PYTHON\folder1\f1.txt
C:\PYTHON\folder1\folder2\f2.txt
C:\PYTHON\vkladena\vkladfil.txt
```

Файл на рівень вище. Перед назвою файлу вказують дві крапки й зворотний слеш (`"..\"`).

Приклад 8.4.

Розмістимо файл upfile.py з кодом у папці на шляху: r"C:\PYTHON\mypack\"

Код у файлі `upfile.py`

```
import os.path
p=os.path.abspath(r"..\proba.txt")
print(p)
f=open(p)
print(f.read())
f.close()
```

Результат:

```
C:\proba.txt
We have got an access to the proba.txt file
```

Нехай дана структура файлів:

```
C:\PYTHON\
    folder1\
      f1.txt
      folder2\
        f2. Txt
        vkladena\
           vklad.txt
```

Також існує копія цієї структури

```
C:\ folder1\
       f1.txt
       folder2\
         f2. Txt
         vkladena\
            vklad.txt
```

4. На початку шляху розташований слеш. Шлях будують від кореня диска. У цьому випадку місце розташування поточного робочого каталогу не має значення.

Приклад 8.5.

```
print(os.path.abspath(r"folder1\f1.txt"))
print(os.path.abspath(r" \folder1\f1.txt"))
print(os.path.abspath(r"folder1\folder2\f2.txt"))
print(os.path.abspath(r"\folder1\folder2\2.txt"))
```

Результат:

```
C:\PYTHON\folder1\file.txt
C:\folder1\file.txt
C:\PYTHON\folder1\folder2\f2.txt
C:\folder1\folder2\f2.txt
```

8.1.4. Прямі та зворотні слеші

В абсолютному й відносному шляхах допустимо використання як прямих, так і зворотних слешів.

Вони будуть автоматично перетворені з урахуванням значення атрибута sep з модуля **os.path**.

Значення цього атрибута залежить від використовуваної операційної системи.

Виведемо значення атрибута sep в операційній системі Windows:

Приклад 8.6.

```
import os.path
print("Вміст атрибуту Sep:",repr(os.path.sep))
# Слеші будуть змінені
```

```
p1=os.path.abspath(r"folder1/f1.txt")
print(repr(p1))
print(p1)
# Встановлено шлях від диску, оскільки слеш на початку
p2=os.path.abspath(r"/folder1/folder2/vkladena/vklad.txt")
print(repr(p2))
print(p2)
```

Результат:

```
    Вміст атрибуту Sep: '\\'
'C:\\PYTHON\\folder1\\f1.txt'
C:\PYTHON\folder1\f1.txt
'C:\\folder1\\folder2\\vkladena\\vklad.txt'
C:\folder1\folder2\vkladena\vklad.txt
```

8.1.5. Особливості використання відносного шляху

При використанні відносного шляху необхідно враховувати місце розташування поточного робочого каталогу. Робочий каталог не завжди збігається з каталогом, у якому перебуває файл, що виконується. Якщо файл запускається за допомогою подвійного клацання на його значку, то каталоги будуть збігатися. Якщо ж файл запускається з командного рядка, то поточним робочим каталогом буде каталог, в якому ми знаходимося під час запуску файлу.

Нехай дана структура файлів:

```
C:\PYTHON\
    first.py
    first1.py
    mypack\
        __init__.py
        lecture25.py
        upfile.py
```

Файл `C:\PYTHON\mypack\_init_.py` створюємо порожнім. Як ви вже знаєте, цей файл указує інтерпретаторові Python, що даний каталог є пакетом з модулями.

Приклад 8.7. Вміст файлу `C:\PYTHON\first.py`

```
import os, sys
import mypack.lecture25 as m
print("%-25s %s" % ("Файл:", os.path.abspath(__file__)))
print("%-25s %s" % ("Поточний робочий каталог:", os.getcwd()))
print("%-25s %s" % ("Каталог модуля:", sys.path[0]))
print("-" * 40)
m.get_data ()
```

Вміст файлу **`C:\PYTHON\mypack\lecture25.py`** наведений в прикладі 8.8.

Приклад 8.8.

```
import os, sys
def get_data():
    print("%-25s%s" % ("Файл:", os.path.abspath(__file__ )))
    print("%-25s%s" % ("Поточний робочий каталог:", os.getcwd()))
    print("%-25s%s" % ("Каталог модуля", sys.path[0]))
```

Запускаємо командний рядок, переходимо в каталог cd C:\PYTHON і запускаємо файл **`first.py`**:

```
C:\PYTHON> python first.py

Файл:                    C:\PYTHON\ first.py Поточний
робочий каталог:C:\PYTHON
Каталог модуля:          C:\PYTHON
----------------------------------------
Файл:      C:\PYTHON\mypack\lecture25.py
Поточний робочий каталог: C:\PYTHON
Каталог модуля:           C:\PYTHON
```

У цьому прикладі поточний робочий каталог збігається з каталогом, у якому розташований файл **`first.py`**.

Однак зверніть увагу на поточний робочий каталог всередині модуля **lecture25.py**.

Якщо всередині цього модуля використати його власний атрибут **__file__** ,то пошук файлу буде зроблений у каталозі.

C:\PYTHON\mypack а не **C:\PYTHON**.

Тепер перейдемо в корінь диска C: командою cd\ і знову запустимо файл first.py:

```
C:\>python C:\PYTHON\first.py

Файл:                    C:\PYTHON\ first.py
Поточний робочий каталог:C:\
Каталог модуля:          C:\PYTHON

----------------------------------------

Файл:          C:\PYTHON\mypack\lecture25.py
Поточний робочий каталог: C:\
Каталог модуля:            C:\PYTHON
```

У цьому випадку поточний робочий каталог не збігається з каталогом, у якому розташований файл first.py.

Для того, щоб каталог з файлом зробити поточним робочим каталогом необхідно використати функцію зміни робочого каталогу **chdir()** з модуля os.

Для прикладу розглянемо вміст файлу **first1.py,** який знаходиться в каталозі C:\PYTHON і запустимо його з каталогу C:\

Приклад 8.9.

```
import os, sys
# Робимо каталог з файлом, що виконується, поточним
os.chdir(os.path.dirname (os.path.abspath(__file__)))
print("%-25s%s" % ("Файл:", __file__ ) )
print("%-25s%s" % ("Поточний робочий каталог:", os.getcwd()))
print("%-25s%s" % ("Каталог модуля:", sys.path[0]))
```

```
import mypack.lecture25 as m
m.get_data ()
```

Ми передаємо значення атрибута **__file__** у функцію **abspath()** з модуля **os.path** і отримуємо абсолютний шлях до файлу з іменем самого файлу

Далі ми «витягаємо» шлях (без назви файлу) за допомогою функції `dirname()` і передаємо його функції **chdir().**

Тепер, якщо у функції **open()** указати назву файлу без шляху, то пошук буде проводитися в каталозі з цим файлом.

Приклад 8.10. Запустимо **chng.py** з каталогу **C:**

```
import os.path, sys
p=os.path.abspath(__file__)
print("Абсолютний шлях до файлу:",p)
d=os.path.dirname(p)
print("Каталог файлу:",d)
print("Наш робочий каталог:", os.getcwd())
os.chdir(d)
print("Тепер це робочий каталог:", os.getcwd())
```

Основний висновок

1. Поточним робочим каталогом буде каталог, з якого запускається файл, а не каталог, у якому розташований файл, що виконується.

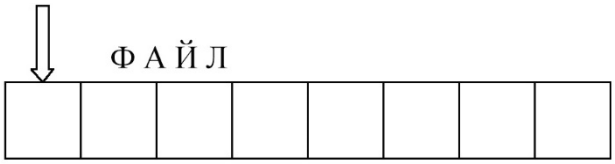
2. Використавши **chdir(),** можемо змінити поточний каталог, вказавши бажаний шлях до робочого каталогу, як параметр функції.

3. Шляхи пошуку файлів не мають жодного відношення до шляхів пошуку модулів.

8.1.6. Параметр **mode** *у функції* **open()**

open(<~~Шлях до файлу~~ >**[, mode='r'] [, buffering=-1] [, encoding=None] [,errors =None] [, newline=None] [, closefd=True]]**)

Необов'язковий параметр **mode** у функції **open()** може набувати наступних значень:



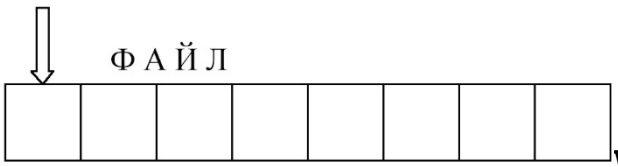
r, r+

r – тільки читання (значення за замовчуванням). *(read)*

Після відкриття файлу покажчик установлюється на початок файлу. Якщо файл не існує, виконується виключення `FileNotFoundError;`

r+ – читання й запис. *(read)*

Після відкриття файлу покажчик установлюється на початок файлу. Якщо файл не існує, то виконується виключення `FileNotFoundError;`



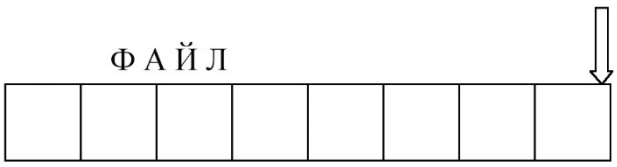
w,w+

w – запис. *(write)*

Якщо файл не існує, він буде створений. Якщо файл існує, він буде перезаписаний. Після відкриття файлу покажчик установлюється на початок файлу;

w+ – читання й запис. *(write)*

Якщо файл не існує, він буде створений. Якщо файл існує, він буде перезаписаний. Після відкриття файлу покажчик установлюється на початок файлу;



a,a+

a – запис. *(add)*

Якщо файл не існує, він буде створений.

Запис здійснюється в кінець файлу.

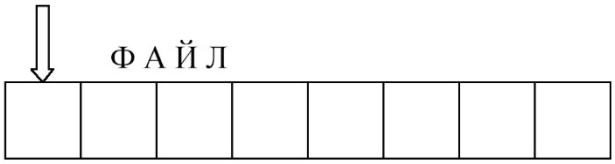
Вміст файлу не видаляється;

a+ – читання й запис. *(add)*

Якщо файл не існує, він буде створений.

Запис здійснюється в кінець файлу.

Вміст файлу не видаляється;



x,x+

x – створення файлу для запису. *(exclusive)*

Якщо файл **уже** існує, виконується виключення `FileExistsError`;

x+ – створення файлу для запису й читання. *(exclusive)*

Якщо файл **уже** існує, виконується виключення `FileExistsError`.

Модифікатори режиму (читання - запис)

Після вказівки режиму може слідувати модифікатор:

b – файл буде відкритий у бінарному режимі. *(binary)*

Файлові методи приймають і повертають об'єкти типу **bytes**;

t – файл буде відкритий у текстовому режимі (значення за замовчуванням у Windows). *(text)*

Можливі модифікації режимів читання-запису:

r,rt,rb; читання r+,rt+,rb+; читання-запис

w,wt,wb, запис w+,wt+,wb+; запис-читання

a,at,ab; запис a+,at+,ab+; запис читання

x,xt,xb, ств.зап. x+,xt+,xb+; ств.зап.-чит.

Файлові методи запису та зчитування рядків

Файлові методи обробляють об'єкти типу **str.**

У цьому режимі буде автоматично виконуватися обробка символу кінця рядка.

Для прикладу створимо файл **file.txt** і запишемо в нього два рядки:

Приклад 8.11.

```
import time
f= open(r"file.txt","w")# Відкриваємо файл на запис
a=f.write("string1\nstring2")# Записуємо два рядки у файл
f.write("\n"+time.ctime())# Записуємо дату
print(a)
f.close()# Закриваємо файл
```

Оскільки ми вказали режим **w**, то:

-якщо файл не існує, він буде створений,

-якщо існує, то буде перезаписаний.

```
# Читаємо файл у текстовому режимі
f= open(r"file.txt","r")#Відкриваємо файл на читання
b=f.read()# Читаємо
print(b)
f.close()
```

Результат

```
string1
string2
Thu Dec  2 08:49:28 2021
```

Тепер виведемо вміст файлу в бінарному й текстовому режимах:

```
import time
f= open(r"file.txt","rb")# Відкриваємо файл на запис
a=f.read()# Записуємо два рядки у файл
f.write("\n"+time.ctime())# Записуємо дату
print(a)
f.close()# Закриваємо файл
```

Результат

```
b'string1\r\nstring2\r\nThu Dec  2 08:49:28 2021'
```

Приклад 8.12.

```
>>> # Бінарний режим (символ \r залишається)
>>> # менеджер контексту with…as
>>> with open(r"file.txt", "rb") as f:
        for line in f:
            print(repr(line))
b'string1\r\n'
b'string2\r\n'
b'Sun Dec 13 14:59:46 2020'
>>> #Текстовий режим (символ \r видаляється)
>>> with open(r"file.txt", "r") as f:
        for line in f:
```

```
            print(repr(line))
```

```
'string1\n'
'string2\n'
'Sun Dec 13 14:59:46 2020'
```

8.1.7. Буферизація запису в файл (параметр buffering)

open(<Шлях до файлу>[, mode='r'] [, **buffering**=-1] [, encoding=None] [,errors =None] [, newline=None]**)**

Для прискорення роботи проводиться буферизація записуваних даних.

Інформація з буфера записується у файл повністю тільки в момент закриття файлу або після виклику функції або методу **flush().**

Керування буферизацією відбувається через параметр **buffering.**

У необов'язковому параметрі buffering можна вказати розмір буфера.

buffering = 0, то дані будуть відразу записуватися у файл (значення припустиме тільки в бінарному режимі).

buffering =1 використовується при по-рядковому записі у файл (значення припустиме тільки в текстовому режимі),

buffering =N, де N - додатне число

Задає приблизний розмір буфера,

buffering = -N від'ємне значення (або відсутність значення)

Задає установку розміру, застосовуваного в системі за замовчуванням. За замовчуванням текстові файли буферизуются по-рядково, а бінарні – частинами, розмір яких інтерпретатор вибирає самостійно в діапазоні від 4096 до 8192 байтів.

8.1.8. Кодування запису в файл (параметр encoding)

open(<Шлях до файлу >[, mode='r'] [, buffering=-1] [, **encoding=None**] [,errors =None] [, newline=None]**)**

Цей параметр застосовуємо тільки за умови текстового режиму відкриття файлу.

При читанні проводиться спроба перетворити дані в кодування Unicode.

При записі виконується зворотна операція – рядок перетвориться в послідовність байтів у визначеному кодуванні.

За замовчуванням призначається кодування, застосовуване в системі. Якщо перетворення неможливе, то виконується виключення. Указати кодування, яке буде використовуватися при записі й читанні файлу, дозволяє параметр **encoding.**

Для прикладу запишемо дані в кодуванні **UTF-8**:

Приклад 8.13.

```
>>> f = open(r"file.txt", "w", encoding="utf-8")
>>> f.write("Рядок") # Записуємо рядок у файл
>>> f.close() # Закриваємо файл
```

Для читання цього файлу слід явно вказати кодування при відкритті файлу:

Приклад 8.14.

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(line)
Рядок
```

Особливості роботи з кодуванням UTF

При роботі з файлами в кодуваннях UTF-8, UTF-16 і UTF-32 слід враховувати, що на початку файлу можуть бути наявними службові символи, називані скорочено BOM (Byte Order Mark, мітка порядку байтів).

Для кодування UTF-8 ці символи є необов'язковими, і в попередньому прикладі вони не були додані у файл при записі.

Щоб символи були додані, у параметрі **encoding** слід указати значення **utf-8-sig**. Запишемо рядок у файл у кодуванні UTF-8 з BOM:

Приклад 8.15.

```
>>> f = open (r"file.txt", "w", encoding="utf-8-sig")
>>> f.write("Рядок") # Записуємо рядок у файл
>>> f.close () # Закриваємо файл
```

Прочитаємо файл з різними значеннями в параметрі **encoding**:

Приклад 8.16.

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(repr(line))
```

```
'\ufeffРядок'
>>> with open(r"file.txt", "r", encoding="utf-8-sig") as f:
        for line in f:
            print(repr(line))
'Рядок'
```

У першому прикладі ми вказали значення utf-8, тому маркер BOM був прочитаний з файлу разом з даними.

У другому прикладі вказано значення utf-8-sig, тому маркер BOM не потрапив в результат.

Якщо невідомо, чи є маркер у файлі, і необхідно отримати дані без маркера, то слід завжди вказувати значення utf-8-sig при читанні файлу в кодуванні UTF-8.

Для кодувань UTF-16 і UTF-32 маркер БОМ є обов'язковим. При вказівці значень utf-16 і utf-32 в параметрі encoding обробка маркера проводиться автоматично.

При запису даних маркер автоматично вставляється в початок файлу, а при читанні він не потрапляє в результат.

Запишемо рядок в файл, а потім прочитаємо його з файлу:

Приклад 8.17.

```
>>> with open(r"file.txt", "w", encoding = "utf-16") as f:
    f.write ( "Рядок")
5
>>> with open (r"file.txt", "r", encoding = "utf-16") as f:
        for line in f:
            print (line)
Рядок
```

8.1.9.Параметр **errors**

open(<Шлях до файлу >[, mode='r'] [, buffering=-1] [, encoding=None] [,**errors =None**] [, newline=None]**)**

У параметрі **errors** можна вказати рівень обробки помилок. Можливі значення:

"strict" (при помилці виконується виключення ValueError – значення за замовчуванням),

"replace" (невідомий символ заміняється символом знак питання або символом з кодом \ufffd),

"ignore" (невідомі символи ігноруються),

"XmlCharRefReplace" (невідомий символ заміняється послідовністю &#xxxx;) і "backslashreplace" (невідомий символ заміняється послідовністю \uxxxx).

8.1.10. Параметр **newline**

open(<Шлях до файлу >[, mode='r'] [, buffering=-1] [, encoding=None] [,errors =None] [, **newline=None**]**)**

Параметр newline задає режим обробки символів кінця рядків.

Підтримувані ним значення такі:

• None (значення за замовчуванням) – виконується стандартна обробка символів кінця рядка. Наприклад, в Windows при читанні символи \r\n перетворяться в символ \n, а при записі проводиться зворотне перетворення;

• "" (порожній рядок) – обробка символів кінця рядка не виконується;

• "<Спеціальний символ>" – зазначений спеціальний символ використовується для позначення кінця рядка, і ніяка додаткова обробка не виконується. Як спеціальний символ можна вказати лише \r\n, \r і \n.

Метод **close()**

Метод **close()** призначений для закривання файлу

```
f=open("file.txt",mode="w")
f.close()
```

Інтерпретатор автоматично видаляє об'єкт, коли на нього відсутні посилання. Тому у невеликих програмах можна явно не закривати файл. Явне закриття файлу є ознакою гарного стилю програм.

Після відкриття файлу функція open() повертає об'єкт, за допомогою якого проводиться подальша робота з файлом.

Тип об'єкта залежить від режиму відкриття файлу і буферизації.

```
f=open("file.txt",mode="w")
print(type(f))
f.close() #Результат <class '_io.TextIOWrapper'>
f=open("file.txt","wb",buffering=0)
print(type(f))
f.close()  # Результат <class '_io.FileIO'>
f=open("file.txt","wb",buffering=-100)
print(type(f))
f.close() # Результат <class '_io.BufferedWriter'>
```

With..as

Мова Python підтримує протокол менеджерів контексту. Використання with..as гарантує закриття файлу незалежно від того, відбулося виключення всередині блоку коду чи ні.

Приклад 8.18.

```
with open("pryklad","w") as f:
    f.write("Рядок 1 прикладу")# Записуємо рядок у файл
f.write("Рядок 2 прикладу")# Тут файл уже закритий автоматично
Traceback (most recent call last):
  File "C:/MYPYTHON/Lexture25/Ex1.py", line 4, in <module>
    f.write("Рядок 2 прикладу")
ValueError: I/O operation on closed file.
```

8.2. Методи зчитування та запису у файли

8.2.1.Метод `write()`

f.write (<дані>) – записує дані у файл.

<дані> - це рядок або послідовність байтів

Для запису тексту необхідно відкрити файл у текстовому режимі.Для запису послідовності байтів необхідно відкрити файл у бінарному режимі.

Пам’ятайте, що не можна записувати рядок у бінарному режимі та послідовність байтів у текстовому режимі.

Метод повертає кількість записаних символів або байтів.

Приклад 8.19. Приклад запису у файл

```
# Текстовий режим
f=open(r"file1.txt","w",encoding="cp1251")
f.write("КПІ - найкращий ЗВО")
f.write(" It is the best")
f.close()
```

Правила запису у текстовому режимі:

1.В параметрах функції **open()** можна вказати encoding

2. Метод **write** приймає тільки 1 аргумент

```
# Бінарний режим
f = open(r"file2.txt", "wb")
f.write(bytes("КПІ - найкращий ЗВО", "cp1251"))
f.write(bytearray(" It is the best", encoding="cp1251"))
f.close()
```

Правило запису у бінарному режимі:

1. В параметрах функції `open()` не можна вказати **encoding**

8.2.2. Методи **read**`() i readline()`

Метод `read()` – зчитує всі дані з файлу.

Якщо файл відкритий у текстовому режимі, то повертається рядок.

Якщо файл відкритий у бінарному режимі, то повертається послідовність байтів.

Приклад 8.20.

```
# Текстовий режим
with open(r"file1.txt","r",encoding="cp1251") as f:
    print(f.read())
```

Результат:

КПІ - найкращий ЗВО It is the best

```
with open(r"file2.txt", "rb") as f:  # Бінарний режим
     dat=f.read()
print(dat)
print(str(dat,"cp1251"))
```

Результат:

b'\xca\xcf\xb2 - \xed\xe0\xe9\xea\xf0\xe0\xf9\xe8\xe9 \xc2\xcd\xc7 It is the best'

КПІ - найкращий ЗВО It is the best

Метод read(<кількість>)

Якщо в методі **read** як параметр вказати число, то за кожний виклик буде повертатися зазначена кількість символів або байтів. Коли досягнуто кінець файлу, метод повертає порожній рядок.

Приклад 8.21.

```
# Текстовий режим
with open(r"file1.txt", "r") as f:
        a=f.read(3)
        b=f.read(16)
print(a, b, sep="\n")
```

Результат роботи:

```
КПІ
 - найкращий ВНЗ
```

Метод readline() – зчитує з файлу один рядок при кожному виклику.

Якщо файл відкритий у текстовому режимі, то повертається рядок, а якщо в бінарному – послідовність байтів. Рядок, що повертається, включає символ переводу рядка. При досягненні кінця файлу повертається порожній рядок.

Приклад 8.22.

```
# Текстовий режим
f=open(r"file3.txt","w",encoding="cp1251")
f.write("КПІ - найкращий ЗВО\n")
f.write("It is the best")
f.close()
f = open(r"file3.txt", "r")
print(f.readline()+f.readline())
f.close()
```

Результат роботи:

КПІ – найкращий ЗВО

It is the best

Метод **readline()** *з параметром*

Метод readline(<Кількість>)

Якщо в параметрі зазначене число, то читаємо:

- до зазначеної кількості символів.

- до символу нового кінця рядка (**\n**),

- до символу кінця файлу,

Якщо кількість символів у рядку менша за зазначену в параметрі, то буде прочитаний один рядок, а не зазначена кількість символів,

```
Symbol\n – 6 символів readline(10)
```

Якщо кількість символів у рядку більша, то повертається зазначена кількість символів.

```
  Symbol\n – 6 символів readline(3)
```

Приклад 8.23.

```
a="КПІ - найкращий ЗВО\n"
b="ФІОТ - найкращий в КПІ\n"
c="ОТ - найкраща кафедра\n"
d="Іванов - найкращий студент"
f=open("file3.txt","w",encoding="cp1251")
f.write(a+b+c+d)
f.close()
f = open ("file3.txt", "r", encoding="cp1251")
print("%s | %s | %s" %  (f.readline(3),f.readline(13),f.readline(4)))
print ("%s%s%s" % (f.readline(23),f.readline(23),f.readline(28)))
```

Результат:

```
КПІ |  - найкращий  | ЗВО
ФІОТ - найкращий в КПІ
ОТ - найкраща кафедра
Іванов - найкращий студент
```

.2.3. Метод **writable()** *і* **writelines()**

Метод **writable()** - повертає True або False

Якщо файл підтримує запис - True,

Якщо не підтримує запис -False

Приклад 8.24.

```
f = open(r"file1.txt", "r")
print("Перевірка можливості запису:",f.writable())

f = open(r"file1.txt", "w")
print("Перевірка можливості запису:",f.writable())
```

Результат:

```
Перевірка можливості запису: False
Перевірка можливості запису: True
```

Метод **writelines()**

Метод writelines(<список>) – записує список у файл.

Варіант1: Список складається з рядків

Якщо всі елементи послідовності є рядками, то файл повинен бути відкритий у текстовому режимі.

Варіант2: Список складається з байтів

Якщо всі елементи є послідовностями байтів, то файл повинен бути відкритий у бінарному режимі.

Приклад 8.25. Приклад запису елементів списку:

```
# Текстовий режим
f = open(r"file4.txt", "w+", encoding="cp1251")
f.writelines(["Рядок1\n", "Рядок2"])
f.close()

a=["Кафедра ОТ\n ","Групи ІП\n ", "Групи ІВ\n ", "Групи ІО"]
f = open(r"file5.txt", "w+", encoding="cp1251")
if f.writable():
    f.writelines(a)
```

```
f.close()
# Бінарний режим
f = open(r"file6.txt", "wb")
arr = [bytes("Рядок1\n", "cp1251"), bytes("Рядок2", "cp1251")]
f.writelines(arr)
f.close()
```

8.2.4. Метод `readlines()`

Метод `readlines()` – зчитує весь вміст файлу в список.

Кожний елемент списку буде містити один рядок, включаючи символ переводу рядка. Виключенням є останній рядок. Якщо він не завершується символом переводу рядка, то символ переводу рядка доданий не буде.

Якщо файл відкритий у текстовому режимі, то повертається список рядків, а якщо в бінарному – список об'єктів типу **`bytes`**.

Приклад 8.26.

```
#Запис через write
f = open (r"file1.txt", "w", encoding="cp1251")
f.write("Рядок1\nРядок2")
f.close()
f = open (r"file1.txt", "r", encoding="cp1251")
print(f.readlines())
f.close()
f = open (r"file1.txt", "r", encoding="cp1251")
print(f.read())
f.close()
['Рядок1\n', 'Рядок2']
Рядок1
Рядок2
#Запис через writelines
f = open(r"file5.txt", "w+", encoding="cp1251")
f.writelines(["Рядок1\n", "Рядок2"])
f.close()
```

```
f = open (r"file5.txt", "r", encoding="cp1251")
print(f.readlines())
f.close()
f = open (r"file5.txt", "r", encoding="cp1251")
print(f.read())
f.close()
['Рядок1\n', 'Рядок2']
Рядок1
Рядок2
```

8.2.5. Перебір файлу `з __next__ ()` *і оператором* `for`

Метод `__next__()` – зчитує один рядок при кожному виклику.

Якщо файл відкритий у текстовому режимі, повертається рядок, а якщо в бінарному – послідовність байтів. При досягненні кінця файлу виконується виключення **`StopIteration`**.

Приклад 8.27.

```
f = open(r"file.txt", "r",encoding="cp1251")
print(f.__next__())
print(f.__next__())
f.close()
# Текстовий режим
f = open(r"file1.txt","r",encoding="cp1251")
print(f.__next__())
print(f.__next__())
print(f.__next__())
f.close()
Результат выполнения:
Рядок1
Рядок2
Traceback (most recent call last):
  File "C:/PYTHON/first.py", line 5, in <module>
    print(f.__next__())    # Досягнутий кінець файлу
StopIteration
```

Перебір файлу по рядках **for**

Ми можемо перебирати файл построчно за допомогою циклу **for**. Цикл **for** на кожній ітерації дає можливість зчитувати один рядок. Для прикладу виведемо всі рядки, попередньо вилучивши символ переводу рядка:

Приклад 8.28.

```
f = open(r"file1.txt","r",encoding="cp1251")
for line in f:
    print(line.rstrip(),end="! ")
f.close ()
```

Результат

Рядок1! Рядок2!

8.3. Сервісні методи роботи з файлами

8.3.1. Методи **flush()** *та* **fileno()**

`flush()` – примусово записує дані з буфера на диск;

`fileno()` – повертає цілочисельний дескриптор файлу.

Значення, що повертається, завжди буде більшим за число 2.

Число 0 закріплене за стандартним введенням **stdin**,

Число 1 – за стандартним виводом **stdout**,

Число 2 – за стандартним виводом повідомлень про помилки **stderr**.

Приклад 8.29.

```
f= open(r"file1.txt","r", encoding="cp1251")
f.flush()
print("Дескриптор =",f.fileno ())
f.close ()
```

Результат виконання:

```
Дескриптор = 3
```

8.3.2. Метод **truncate ([<Кількість>])**

```
truncate ([<Кількість>])
```

1. Обрізає файл до зазначеної кількості символів

(якщо заданий текстовий режим)

2. Обрізає файл до зазначеної кількості байтів

(у випадку бінарного режиму).

3. При обрахуванні довжини файлу враховує символи кінця рядка \r\n

4. Метод повертає новий розмір файлу.

5. Якщо величина параметру перевищує довжину файла залежно від платформи може включати або не включати «EOF».

Приклад 8.30.

```
#Записуємо в файл
f = open (r"file1.txt", "w", encoding="cp1251")
f.write("Рядок1\nРядок2")
f.close()
# Зчитуємо з файлу
f = open (r"file1.txt", "r+", encoding="cp1251")
print(f.read())
f.truncate(11)
f .close ()
print("Після урізання отримали")
with open(r"file1.txt", "r", encoding="cp1251") as f:
  print(f. read ())
```

Результат виконання:

```
Рядок1
Рядок2
Після урізання отримали
Рядок1
Ряд
Маємо 9 символів +\r\n
```

8.3.3. Метод `tell()`

`tell()` – повертає позицію покажчика відносно початку файлу у вигляді цілого числа.

\r - додатковий байт, хоча цей символ видаляється при відкритті файлу в текстовому режимі.

Приклад 8.31.

```
with open(r"file.txt", "w", encoding="cp1251") as f:
   f.write("String1\nString2")
f = open(r"file.txt", "r", encoding="cp1251")
print("Поточна позиція:",f.tell()) # Покажчик на початку файлу
print(f.readline()) # Переміщуємо покажчик
print("Поточна позиція:",f.tell()) # Повертає 9 (7 + '\r\n'), а не 7
f.close()
```

Результат виконання:

```
Поточна позиція: 0
String1
Поточна позиція: 9
```

Як відобразити символ **\r**

Щоб уникнути даної невідповідності, слід відкривати файл у бінарному режимі, а не в текстовому. У цьому випадку бачимо всі байти, які записані у файл

Приклад 8.32.

```
with open(r"file.txt", "w", encoding="cp1251") as f:
   f.write("String1\nString2")
f = open(r"file.txt", "rb")
print(f.readline()) # Переміщуємо покажчик
print("Поточна позиція:",f.tell())
f.close()
Результат виконання:
b'String1\r\n'
Поточна позиція: 9
```

8.3.4. Метод **seek**() і **seekable()**

seek(<Зсув>[,<Позиція>]) – установлює покажчик у позицію, що має зсув <Зсув> відносно позиції <Позиція>.

У параметрі <Позиція> можуть бути зазначені наступні атрибути з модуля io або відповідні їм значення:

• io.SEEK_SET або **0** – початок файлу (значення за замовчуванням);

• io.SEEK_CUR або **1**– поточна позиція покажчика. Додатне значення зсуву викликає переміщення у напрямку кінця файлу, від'ємне – у напрямку початку;

• io.SEEK_END або **2** – переміщення від кінця файлу.

Виведемо значення цих атрибутів:

```
>>> import io
>>> io.SEEK_SET, io.SEEK_CUR, io.SEEK_END
(0, 1, 2)
```

Приклад використання методу **seek() :**

Приклад 8.33.

```
import io
f = open(r"file.txt", "rb")
print("Зсунути на ",f.seek(9, io.SEEK_CUR)) # 9 байт від покажчика
print("Поточна позиція:",f.tell())
print("На початок: ",f.seek(0,io.SEEK_SET)) # Зсуваємо на початок
print("Поточна позиція:",f.tell())
print("-9 від кінця",f .seek(-1, io. SEEK_END)) # -1 байтов від кінця
print("Поточна позиція:",f.tell())
f .close ()
```

Результат виконання:

```
Зсунути на  9
Поточна позиція: 9
На початок:  0
Поточна позиція: 0
-1 від кінця 15
Поточна позиція: 15
```

Метод **seekable()**

seekable() – повертає **True**, якщо покажчик файлу можна зсунути в іншу позицію, і **False –** якщо ні:

Приклад 8.34.

```
import io
f= open(r"file.txt", "rb")
print(f. seekable())
if f. seekable():
    print("7 від початку",f .seek(7, io.SEEK_CUR))#7
print("Поточна позиція:",f.tell()) #7
    print("Поточна позиція:",f.read())
f .close ()
```

Результат

```
True
7 від початку 7
Поточна позиція: b'\r\nString2'
```

8.3.5. Атрибути файлового об'єкта

Файловий об'єкт підтримує атрибути, які дозволяють дізнатися, про його поточний стан.

1.Атрибут name містить ім'я відкритого файлу;

2. Атрибут mode – зберігає режим, у якому був відкритий поточний файл, наприклад: **"rb"- бінарний режим для читання**

3. Атрибут closed повертає True, якщо файл був закритий, і False – якщо ні.

4. Атрибут encoding містить кодування, у якому був відкритий текстовий файл

5. Атрибут **buffer** забезпечує доступ до файлу у тестовому режимі

Приклад застосування атрибутів файлу **name, mode і closed**

Приклад 8.35

```
import os
f = open(r"file.txt", "rb+")
print("Им'я файлу:",f.name) # Ім'я файла даних
print("Режим відкривания:",f.mode)
print("До закриття",f.closed)
f.close()
```

```
print("Після закриття",f.closed)
```

Результат виконання:

```
Ім'я файлу: file.txt
Режим відкривання: rb+
До закриття False
Після закриття True
```

Атрибут **encoding**

encoding – назва кодування, яке буде використовуватися для перетворення рядків перед записом у файл або при читанні. Атрибут доступний тільки в текстовому режимі.

Змінити значення атрибута не можна, оскільки він доступний тільки для читання.

Приклад 8.36.

```
f = open(r"file.txt", "a", encoding="cp1251")
print("Кодування: ",f.encoding)
f.close()
```

Результат виконання: Кодування: cp1251

Застосування **encoding** *у* **stdout**

Стандартний вивід **stdout** також є файловим об'єктом.

Атрибут **encoding** цього об'єкта завжди містить кодування пристрою виводу, тому рядок перетвориться в послідовність байтів у правильному кодуванні.

Наприклад, при запуску у вікні редактора Pycharm – значення "UTF-8".

Приклад 8.37.

```
import sys
# Кодування файла стандартного виводу stdout
print("Кодування STDOUT =",sys.stdout.encoding)
```

Результат

```
Кодування STDOUT = UTF-8
```

Атрибут **buffer**

buffer – дозволяє одержати доступ до буфера. Атрибут доступний тільки в текстовому режимі. За допомогою цього атрибуту можна записати послідовність байтів у текстовий потік.

Приклад 8.38.

```
#Приклад застосування атрибута buffer
f = open(r"file.txt", "w", encoding="cp1251")
print(f.buffer.write(bytes("Рядок","cp1251")))
#Повертає кількість символів
f.close()
with open(r"file.txt","r",encoding="cp1251") as f:
   print(f. read ())
```

Результат

```
5
Рядок
```

Функції для маніпулювання файлами **shutil**

Для копіювання й переміщення файлів призначені наступні функції з модуля **shutil**:

copyfile(<Початковий файл>,<Куди копіюємо>)

Функція дозволяє скопіювати вміст файлу в інший файл. Ніякі метадані та права доступу не копіюються. Якщо файл існує, то він буде перезаписаний. Якщо файл не вдалося скопіювати, виконується виключення **OsError** або одне з виключень, що є підкласом цього класу.

Приклад 8.39.

```
import shutil # Підключаємо модуль
pt = shutil.copyfile(r"file1.txt", r"file11.txt")
print("Записано в",pt)
#шлях не існує:
shutil.copyfile(r"file1.txt", r"C:\P_NEW\file22.txt")
```

Результат виконання:

```
Записано в file11.txt
FileNotFoundError: [Errno 2] No such file or directory: 'C:\\P_NEW\\file22.txt'
```

Виключення **`FileNotFoundError`** є підкласом класу `OSError` і виконується, якщо зазначений файл не знайдений.

Функція **`copyfile()`** як результат повертає шлях файлу, куди були скопійовані дані.

8.3.6. Функції копіювання і вилучення файлів

Функція `copy()`

`copy(<Копійований файл>, <Куди копіюємо>)`

Функція дозволяє скопіювати файл **разом з правами доступу.** Якщо файл існує, то він буде перезаписаний. Якщо файл не вдалося скопіювати, виконується виключення `OSError` або одне з виключень, що є підкласом цього класу.

Приклад 8.40.

```
#Копіювання з правами доступу
import shutil # Підключаємо модуль
my_path = shutil.copy(r"file1.txt", r"file13.txt")
print("Записано в",my_path)  #Шлях копіювання
```

Функція **`copy()`** як результат повертає шлях скопійованого файлу.

Функція **`copy2()`**

`copy2``(<Копіюємий файл>, <Куди копіюємо>)`

Функція дозволяє скопіювати файл **разом з метаданими.** Якщо файл існує, то він буде перезаписаний. Якщо файл не вдалося скопіювати, виконується виключення `OSError` або одне з виключень, що є підкласом цього класу.

Приклад 8.41.

```
# Копіювання з метаданими
import shutil # Підключаємо модуль
my_path = shutil.copy2(r"file1.txt", r"file15.txt")
print("Записано в ",my_path)#Шлях копіювання
```

Функція `copy2()` як результат повертає шлях скопійованого файлу.

Функція **`move()`**

move(<Шлях до файлу>, <Куди переміщаємо>)

Функція переміщає файл у зазначене місце з видаленням вхідного (початкового) файлу. Якщо файл існує, то він буде перезаписаний. Якщо файл не вдалося перемістити, виконується виключення `OSError` або одне з виключень, що є підкласом цього класу.

Приклад переміщення файлу `file1.txt` у каталог `file15.txt`

Приклад 8.42.

```
#Переміщення
import shutil # Підключаємо модуль
my_path = shutil.move(r"file1.txt", r"file15.txt")
print("Записано в ",my_path)#Шлях переміщення
```

Функція **move()** як результату повертає шлях переміщеного файлу.

Функція **rename()**

rename(<Старе ім'я>, <Нове ім'я>)

Призначена для перейменування файлів. Знаходиться в модулі `os`:

Функція перейменовує файл. Якщо файл не вдалося перейменувати, виконується виключення `Oserror` або одне з виключень, що є підкласом цього класу.

Приклад 8.43.

```
import os # Подключаем модуль
try:
   os.rename(r"file3.txt", "file16.txt")
except OSError:
   print("Файл не вдалося перейменувати")
else:
   print("Файл успішно перейменований")
```

Результат виконання: Файл успішно перейменований

Функції **remove()** *і* **unlink()**

remove (<Шлях до файлу>) unlink (<Шлях до файлу>)

Функції дозволяють вилучити файл. Якщо файл не вдалося вилучити, виконується виключення `OSError` або одне з виключень, що є підкласом цього класу.

Приклад 8.44.

```
import os # Подключаем модуль
f = open(r"file2.txt", "w")
f.write("Рядок1\nРядок2")
f.close()
print ("До remove: %s" %os.listdir(os.getcwd()))
os.remove(r"file2.txt")
print ("Після remove: %s" %os.listdir(os.getcwd()))
До remove: ['.idea', 'file2.txt', 'file4.txt', 'first.py']
Після remove: ['.idea','file4.txt', 'first.py']
```

Приклад 8.45.

```
import os # Подключаем модуль
f = open(r"file4.txt", "w")
f.write("Рядок1\nРядок2")
f.close()
print ("До unlink: %s" %os.listdir(os.getcwd()))
os.unlink(r"file4.txt")
print ("Після unlink: %s" %os.listdir(os.getcwd()))
До unlink: ['.idea', 'file4.txt', 'first.py']
Після unlink: ['.idea', 'first.py']
```

Права доступу до файлів і каталогів

В операційних системах сімейства UNIX кожному об'єкту (файлу або каталогу) призначаються права доступу, надавані тим або іншим різновидам користувачів: власнику, групі й решті інших.

Можуть бути призначені наступні права доступу:

-читання. Користувач може читати файл або каталог

-запис. Користувач може записувати у файл або каталог

-виконання. Користувач може запускати файл на виконання

Позначення прав доступу

`r` – файл можна читати, а вміст каталогу можна переглядати;

`w` – файл можна модифікувати, видаляти й перейменовувати, а в каталозі можна створювати або видаляти файли. Каталог можна перейменувати або вилучити;

`x` – файл можна виконувати, а в каталозі можна виконувати операції над файлами, у тому числі робити в ньому пошук файлів.

Права доступу до файлу визначають записом типу: **`-rw-r--r--`**

Перший символ `(-)` означає, що це файл, і не задає прав доступу.

Далі три символи `(rw-)` задають права доступу для власника: rw означають читання й запис, символ – означає, що права на виконання немає.

Наступні три символи задають права доступу для групи `(r--)` – тільки читання.

Останні три символи `(r--)` задають права для всіх інших користувачів – також тільки читання.

Права доступу до каталогу визначають таким рядком: **`drwxr-xr-x`**

Перша буква `(d)` означає, що це каталог.

Власник може виконувати в каталозі будь-які дії `(rwx)`.

Група може тільки читати й виконувати пошук `(r-x)`.

Інші користувачі також можуть тільки читати й виконувати пошук `(r-x)`.

Для того, щоб каталог можна було переглядати, повинні бути встановлені права на виконання `(x)`.

Права доступу можуть позначатися й числом

Такі числа називаються *маскою прав доступу.* Число містить три цифри.

Кожна цифра може змінюватися від 0 до 7 у двійковій системі числення.

Перша цифра задає права для власника,друга – для групи, третя – для всіх інших користувачів.

Наприклад, права доступу `-rw-r--r--` відповідають числу 110 100 100 ⟶ `644` .

`rw` позначаємо як 6, тобто 110 у двійковій системі `r` позначаємо як 4, тобто 100 у двійковій системі.

Зіставимо числа, що входять у маску прав доступу, з двійковим і буквеним записами в таблиці.

аблиця 8.1.

Права доступу в різних записах

Вісімкова цифра	Двійковий запис	Буквений Запис	Вісімкова цифра	Двійковий запис	Буквений запис
0	000	---	4	100	r--
1	001	--x	5	101	r-x
2	010	-w-	6	110	r-w
3	011	-wx	7	111	rwx

Тепер зрозуміло, що, згідно з даними цієї таблиці, права доступу

`rw-r--r--, 110 100 100,` 644

мають еквівалентні представлення

Таким чином, якщо право надане, то у відповідній позиції має бути 1, а якщо ні – то 0.

Визначення доступу до файлу

Для визначення доступу до файлу або каталогу призначена функція **`access()`** з модуля **`os`**. Функція має наступний формат:

`access`(<**`Шлях`**>, <**`Режим`**>)**.**

Функція повертає `True`, якщо перевірка пройшла успішно, повертає **`False`** якщо доступ відсутній.

У параметрі <**`Режим`**> можуть бути зазначені наступні константи, що визначають тип перевірки:

`os.F_OK` – перевірка наявності шляху або файлу:

`os.R_OK` – перевірка на можливість читання файлу або каталогу;

`os.W_OK` – перевірка на можливість запису у файл або каталог;

`os.X_OK` – визначення, чи є файл або каталог виконуваним.

Приклад 8.46.

```
import os # Підключаємо модуль os
# Перевірка існування файлу
print("file.txt",os.access(r"file.txt", os.F_OK))
# Перевірка існування каталогу
print("C:\PYTHON",os.access(r"C:\PYTHON", os.F_OK))
# каталог не існує
print("C:\PYTHON_NEW",os.access(r"C:\PYTHON_NEW", os.F_OK))
```

Результат виконання:

```
file.txt True
C:\PYTHON True
C:\PYTHON_NEW False
```

Зміна прав доступу

Щоб змінити права доступу з програми, необхідно скористатися функцією **chmod()** з модуля **os**.

Функція має наступний формат:

chmod(<**Шлях**>, <**Права доступу**>)

Права доступу задають у вигляді числа, перед яким слід указати комбінацію символів 0o (це відповідає вісімковому запису числа):

```
>>> os.chmod(r"file.txt", 0o777)
# Повний доступ до файлу
```

Замість числа можна вказати комбінацію констант із модуля **stat**. За додатковою інформацією звертайтеся до документації з модуля.

Функції модуля **os.path**

Модуль os.path містить додаткові функції, що дозволяють перевірити наявність файлу, одержати розмір файлу й ін.

```
exists (<Шлях>)
exists (<Номер файлу (Дескриптор)>)     #f.fileno
```

Функція перевіряє зазначений шлях на існування. Функція повертає True, якщо шлях до файлу існує. Функція повертає False – якщо шлях до файлу не існує

параметр <Шлях> – це послідовність типу str, який використовує зворотні слеші, як роздільники, для запису послідовності дерева папок.

Параметр <**Дескриптор** файлу> – це число, яке повертає метод fileno().

Перевірка існування файлу за шляхом

Приклад 8.47.

```
import os.path
#Зворотні слеші
print("file.txt",os.path.exists(r"file.txt"))

print("file2.txt",os.path.exists(r"file2.txt"))

print("C:\PYTHON",os.path.exists(r"C:\PYTHON"))

print("C:\P_NEW",os.path.exists(r"C:\P_NEW"))
Результат виконання:
file.txt True
file2.txt False
C:\PYTHON True
C:\P_NEW False
```

Перевірка існування файлу за дескриптором

Приклад 8.48.

```
import os.path
f=open(r"C:\student\file.txt","w")
f.write("Запис даних")
a=f.fileno()
print("Дескриптор:",a)
p= os.path.exists(a)
print(r"C:\student\file.txt",p)
f.close()
f=open(r"..\file.txt")
b=f.fileno()
```

```
print("Дескриптор:",b)
print(os.path.abspath(r"..\file.txt"),os.path.exists(b))
f.close()
```

Результат виконання:

```
Дескриптор: 4
C:\student\file.txt True
Дескриптор: 4
C:\file.txt True
```

Застосування модуля **pathlib**

Модуль **pathlib** у Python надає різні класи, що представляють шляхи до файлової системи. Метод **pathlib.Path.exists()** перевіряє, чи існує даний шлях до файлу або каталогу.

Приклад 8.49.

```
# Імпортуємо клас Path
from pathlib import Path
path = 'C:/PYTHON/file.txt'  #Прямі слеші
dpath= 'C:/PYTHON'
# Створюємо об'єкт класу Path
obj = Path(path)
dobj = Path(dpath)
# Перевіряємо наявність файлу та каталогу
print(obj.exists(),dobj.exists())
```

Результат **True True**

Цей метод повертає **True**, якщо шлях існує, інакше повертає **False**.

8.3.7. Функції для визначення параметрів файлів

Функція **getsize()**

Формат функції: `getsize (<Шлях до файлу>)`

Функція повертає розмір файлу в байтах. Якщо файл не існує, виконується виключення `FileNotFoundError`:

Приклад 8.50.

```
import os.path
name=r"file.txt"
name1=r"file2.txt"
try:
 a=os.path.getsize(name)#Файл існує
 print("Розмір файлу {0} = {1}".format(name,a))
except FileNotFoundError:
    print("Не вдається знайти файл "+name)
try:
 os. path. getsize(name1)
 print("Розмір файлу {0} = {1}".format(name1,a))
except FileNotFoundError:
    print("Не вдається знайти файл "+name1)
```

Функція **getatime () (access time)**

Формат функції: `getatime (<Шлях до файлу>)`

Початок епохи (00:00:00 UTC) 1 січня 1970 року

Функція служить для визначення часу останнього доступу до файлу. Як значення функція повертає кількість секунд, що пройшли з початку епохи. Якщо файл не існує, виконується виключення **FileNotFoundError**.

Приклад 8.51.

```
import os.path
import time
t = os.path.getatime(r"file.txt")
print(t)
print(time.strftime("%d.%m.%Y %H:%M:%S",time.localtime(t)))
```

Результат

```
1638903598.4259105
07.12.2021 20:59:58
```

Функція getctime **() (**creation time**)**

Формат функції:`getctime (<Шлях до файлу>)`

Функція дозволяє довідатися дату створення файлу. Як значення функція повертає кількість секунд, що пройшли з початку епохи. Якщо файл не існує, виконується виключення **`FileNotFoundError`**.

Приклад 8.52.

```
import os.path
import time
try:
 t = os.path.getctime(r"file.txt")
 print(t)
 print(time.strftime("%d.%m.%Y %H:%M:%S",time.localtime(t)))
except FileNotFoundError:
   print("File is not found")
```

Результат

```
1638427768.9862618
02.12.2021 08:49:28
```

Функція getmtime () (modification time)

Формат функції: `getmtime (<Шлях до файлу>)`

Повертає час останньої зміни файлу. Як значення функція повертає кількість секунд, що пройшли з початку епохи. Якщо файл не існує, виконується виключення **`FileNotFoundError`**.

Приклад 8.53.

```
import os.path
import time
try:
 t = os.path.getmtime(r"file.txt")
 print(t)
 print(time.strftime("%d.%m.%Y %H:%M:%S",time.localtime(t)))
except FileNotFoundError:
   print("File is not found")
```

Результат

1608454311.447067

20.12.2020 10:51:51

Функція **getatime () (access time)** визначення часу останнього доступу до файлу

Функція getctime () (creation time) визначення часу створення файлу

Функція getmtime () (modification time) визначення часу останньої зміни файлу

Функція **stat()** *з модуля* **os**

Формат функції: `stat(<Шлях до файлу>)`

Функція дозволяє одержати параметри файлу в об'єкті `stat_result.`

Об'єкт містить десять атрибутів:

`st_mode`	Тип файлу та режим доступу
`st_ino`	Номер файлу в структурі Айнод (linux),
`st_dev,`	Ідентифікатор приладу зберігання файлу
`st_nlink,`	Визначає місце файлу на диску
`st_uid,`	Ідентифікатор користувача
`st_gid,`	Ідентифікатор групи користувача
`st_size,`	Розмір файлу
`st_atime,`	Час останнього доступу
`st_mtime`	Час останньої модифікації
`st_ctime.`	Час створення

Приклад використання функції **stat()** наведений у прикладі.

Приклад 8.54.

```
import os,time
s = os.stat(r"file.txt")
print("{0:20}    {1:7}".format("Атрибути файлу:", "file.txt" ))
print(s)

print("-------------------------------------------------")
print("{0:<30} {1:<20}".format("Розмір файлу file.txt:",s.st_size ))
tc=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_ctime))
```

```
print("{0:<30} {1:<20}".format("Час створення file.txt:",tc ))
ta=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_atime))
print("{0:<30} {1:<20}".format("Час доступу file.txt:",ta ))
tm=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_mtime))
print("{0:<30} {1:<20}".format("Час модифікації file.txt:",tm ))
print("-------------------------------------------------")
```

Результати виконання:

```
Атрибути файлу:           file.txt
os.stat_result(st_mode=33206, st_ino=46724846133972803,
st_dev=3500846520, st_nlink=1, st_uid=0, st_gid=0,
st_size=14, st_atime=1638903598, st_mtime=1638435299,
st_ctime=1638427768)
-----------------------------------------Розмір файлу
file.txt:         14
Час створення file.txt:         02.12.2021 08:49:28
Час доступу file.txt:           07.12.2021 20:59:58
Час модифікації file.txt:       02.12.2021 10:54:59
-----------------------------------------
```

Функція **utime()** *з модуля* **os**

Оновити час останнього доступу й час зміни файлу дозволяє функція **utime()** з модуля **os**.

utime(<Шлях до файлу>, None)

utime(<Шлях до файлу>,(<Час доступу>, <Час модифікації>))

Параметр `<Шлях до файлу>` це послідовність типу `str`, яка використовує зворотні слеші, як роздільники, для запису послідовності дерева папок.

Параметр 2 = `None`: час доступу й зміни файлу буде поточним.

Параметр 2 - кортеж з нових значень у вигляді кількості секунд, що пройшли з початку епохи.

Якщо файл не існує, виконується виключення `OSError`.

Приклад використання функції **utime()** наведений нижче.

Приклад 8.55.

```
s=os.stat(r"file.txt")
print("{0:20}     {1:7}".format("Атрибути файлу:", "file.txt" ))
print(s)
print("-------------------------------------------------")
ta=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_atime))
print("{0:<30} {1:<20}".format("Час доступу file.txt:",ta ))
tm=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_mtime))
print("{0:<30} {1:<20}".format("Час модифікації file.txt:",tm ))
print("-------------------------------------------------")
os.utime(r"file.txt", None)
s=os.stat(r"file.txt")
print("{0:20} {1:7}".format("Встановлення поточного часу ", "file.txt" ))
print(s)
print("-------------------------------------------------")
ta=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_atime))
print("{0:<30} {1:<20}".format("Час доступу file.txt:",ta ))
tm=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_mtime))
print("{0:<30} {1:<20}".format("Час модифікації file.txt:",tm ))
print("-------------------------------------------------")

t = time.time() - 10000
os.utime(r"file.txt", (t, t))
s=os.stat(r"file.txt")
print("{0:20}     {1:7}".format("Встановити -10000сек:", "file.txt" ))
print(s)
print("-------------------------------------------------")
ta=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_atime))
print("{0:<30} {1:<20}".format("Час доступу file.txt:",ta ))
```

```
tm=time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(s.st_mtime))
print("{0:<30} {1:<20}".format("Час модифікації file.txt:",tm ))
print("---------------------------------------------")

Атрибути файлу:          file1.txt
os.stat_result(st_mode=33206, st_ino=3659174697911814,
st_dev=3500846520, st_nlink=1, st_uid=0, st_gid=0,
st_size=34, st_atime=1638905410, st_mtime=1638776174,
st_ctime=1638905410)
-----------------------------------------
Час доступу file.txt:          07.12.2021 21:30:10
Час модифікації file.txt:      06.12.2021 09:36:14
-----------------------------------------
Встановлення поточного часу  file1.txt
os.stat_result(st_mode=33206, st_ino=3659174697911814,
st_dev=3500846520, st_nlink=1, st_uid=0, st_gid=0,
st_size=34, st_atime=1638905422, st_mtime=1638905422,
st_ctime=1638905410)
-----------------------------------------
Час доступу file.txt:          07.12.2021 21:30:22
Час модифікації file.txt:      07.12.2021 21:30:22
-----------------------------------------
Встановити -10000сек:     file1.txt
os.stat_result(st_mode=33206, st_ino=3659174697911814,
st_dev=3500846520, st_nlink=1, st_uid=0, st_gid=0,
st_size=34, st_atime=1638895422, st_mtime=1638895422,
st_ctime=1638905410)
-----------------------------------------
Час доступу file1.txt:         07.12.2021 18:43:42
Час модифікації file1.txt:     07.12.2021 18:43:42
-----------------------------------------
```

Перетворення шляху до файлу або каталогу

Перетворити шлях до файлу або каталогу дозволяють наступні функції з модуля `os.path`:

Формат функції: `abspath (<Відносний шлях>)`

Функція перетворить відносний шлях в абсолютний, враховуючи місце розташування поточного робочого каталогу.

Приклад 8.56.

```
import os.path
print("{0:<30} {1:<20}".format("Відносний шлях","file.txt"))
p1 = os.path.abspath(r"file.txt")
print("{0:<30} {1:<20}".format("Абс. шлях",p1))
print("-----------------------------------------------")
print("{0:<30} {1:<20}".format("Відносний шлях в підкаталог","folder1\\file.txt"))
p2 = os.path.abspath(r"folder1/file.txt")
print("{0:<30} {1:<20}".format("Абс. шлях в підкаталог",p2))
print("-----------------------------------------------")
print("{0:<30} {1:<20}".format("Відносний шлях в надкаталог","..\\file.txt"))
p3 = os.path.abspath(r" .. /file.txt")
print("{0:<30} {1:<20}".format("Абс. шлях в надкаталог",p3))
print("-----------------------------------------------")
```

Результат роботи

```
Відносний шлях                 file.txt
Абс. шлях                      C:\PYTHON\file.txt
-----------------------------------------
Відносний шлях в підкаталог    folder1\file.txt
Абс. шлях в підкаталог         C:\PYTHON\folder1\file.txt

-----------------------------------------
```

```
Відносний шлях в надкаталог    ..\file.txt
Абс. шлях в надкаталог         C:\file.txt
-----------------------------------------
```

Атрибут **sep** *з модуля* **os.path**

Можна вказувати як прямі, так і зворотні слеші. Вони будуть автоматично перетворені з урахуванням значення атрибута `sep` з модуля `os.path`.

Значення цього атрибута залежить від використовуваної операційної системи.

Виведемо значення атрибута `sep` в операційній системі Windows:

Приклад 8.57.

```
import os.path
a=os.path.sep
print(repr(a), a, sep="=")
b=os.sep
print(repr(b),b,sep="-")
```

Результат:

```
'\\'=\
'\\'-\
```

Форматування зворотних слешів

При вказівці шляху в Windows слід враховувати, що зворотний слеш є спеціальним символом.

1.Тому слеш необхідно подвоювати (екранувати)

2.Замість звичайних рядків використовувати неформатовані рядки з “r”

Приклад 8.58.

```
p1="C:\\PYTHON\\file.txt"   # Правильно
print(repr(p1))
p2= r"C:\PYTHON\file.txt"   # Правильно
print(repr(p2))
p3= "C:\PYTHON\file.txt"   # Неправильно!!!
print(repr(p3))
```

Результат: `'C:\\PYTHON\\file.txt'`

```
'C:\\PYTHON\\file.txt'
'C:\\PYTHON\x0cile.txt'
```

3. Можна користуватися прямими слешами з перетворенням через **os.path.abspath**

Якщо необхідно мати слеш у кінці рядка, то використовуємо звичайні рядки:

Приклад 8.59.

```
import os.path
c = os.path.abspath("C:/PYTHON/file.txt")
print(c)
a=os.path.abspath("C:/temp/new")+"\\"# Правильно
print("{}---{}".format(repr(a),a))
# Можна слеш видалити
b=r"C:\temp\new\\"[:-1]
print("{}---{}".format(repr(b),b))
```

Результат:

```
C:\PYTHON\file.txt
'C:\\temp\\new\\'---C:\temp\new\
'C:\\temp\\new\\'---C:\temp\new\
```

Функція isabs()

Формат функції: isabs(<Шлях>)

True, якщо абсолютний, False, якщо відносний.

Приклад 8.60.

```
import os
a=os.path.isabs(r"C:\PYTHON\file.txt")
print(r"C:\PYTHON\file.txt:",a)
b=os.path.isabs("file.txt")
print("file.txt:", b)
pr=os.path.abspath("file.txt")
c=os.path.isabs(pr)
print(pr,c)
pu=os.path.abspath("..\\file.txt")
```

```
d=os.path.isabs(pu)
print(pu,d)
```

Результат:

```
C:\PYTHON\file.txt: True
file.txt: False
C:\PYTHON\file.txt True
C:\file.txt True
```

Функція **basename()**

Формат функції: **basename(<Шлях>)**

Функція повертає ім'я файлу або папки без шляху до відповідного файлу чи папки:

Приклад 8.61.

```
import os
a = os.path.basename(r"C:\PYTHON\folder1\file.txt")
print("{0}----{1}".format(repr(a),a))

b = os.path.basename(r"C:\PYTHON\folder1")
print("{0}---{1}".format(repr(b),b))

c=os.path.basename("C:\\PYTHON\\folder1\\")
print("{0}---{1}".format(repr(b),b))
```

Результат:

```
'file.txt'----file.txt
'folder1'---folder1
'folder1'---folder1
```

Функція **dirname()**

Формат функції: **dirname(<Шлях>)**

Функція повертає шлях до папки, де зберігається файл:

Приклад 8.62.

```
import os
a=os.path.dirname(r"C:\PYTHON\folder1\file4.txt")
```

```
print("{0}----{1}".format(repr(a),a))
b=os.path.dirname(r"C:\PYTHON\folder")
print("{0}----{1}".format(repr(b),b))
c=os.path.dirname("C:\\PTHON\\folder\\")
print("{0}----{1}".format(repr(c),c))
```

Результат:

'C:\\PYTHON\\folder1'----C:\PYTHON\folder1

'C:\\PYTHON'----C:\PYTHON

'C:\\PTHON\\folder'----C:\PTHON\folder

Функція **split (<Шлях>)**

Формат функції: **split(<Шлях>)**

Функція повертає кортеж із двох елементів: шляху до папки, де зберігається файл, і назви файлу:

Приклад 8.63.

```
import os
a = os.path.split(os.path.abspath("file.txt"))
print("{}".format(a))
b = os.path.split(os.path.abspath("folder1"))
print("{}".format(b))
c = os.path.split("C:\\PYTHON\\folder1\\")
print(c)
```

Результат:

```
('C:\\PYTHON', 'file.txt')
('C:\\PYTHON', 'folder1')
('C:\\PYTHON\\folder1', '')
```

Функція **splitdrive()**

Формат функції: splitdrive (<Шлях>)

Функція повертає кортеж із двох елементів: ім’я диску та відносний шлях до файлу:

Приклад 8.64.

```
import os
a = os.path.splitdrive(os.path.abspath("file.txt"))
```

```
print("{}".format(a))
b = os.path.splitdrive(os.path.abspath("folder1"))
print("{}".format(b))
c = os.path.splitdrive("C:\\PYTHON\\folder1\\")
print(c)
```

Результат:

```
('C:', '\\PYTHON\\file.txt')
('C:', '\\PYTHON\\folder1')
('C:', '\\PYTHON\\folder1\\')
```

Функція **splitext()**

Формат функції: `splitext (<Шлях>)`

Функція повертає кортеж із двох елементів: шляху з назвою файлу, але без розширення, і розширення файлу (фрагмент після останньої крапки):

Приклад 8.65.

```
import os
a = os.path.splitext(os.path.abspath("file.txt"))
print("{}".format(a))
b = os.path.splitext(os.path.abspath("folder1"))
print("{}".format(b))
c = os.path.splitext("C:\\PYTHON\\folder1\\")
print(c)
```

Результат:

```
('C:\\PYTHON\\file', '.txt')
('C:\\PYTHON\\folder1', '')
('C:\\PYTHON\\folder1\\', '')
```

Функція

Формат функції:

`os.path.join(<Шлях1>[, ... , <Шляхn>])`

Функція з'єднує зазначені елементи шляху, за необхідності вставляючи між ними роздільники:

Приклад 8.66.

```
import os
a = os.path.join("C:\\", "PYTHON\\folder1", "file.txt")
print(a)
b = os.path.join("C:/", "MYPYTHON/Lecture27/", "file.txt")
print(b)
c = os.path.join("C:\\", "MYPYTHON\\Lecture27", "file.txt")
print(c)
d = os.path.join("C:\\", "MYPYTHON/Lecture27", "file.txt")
print(d)
```

Результат

```
C:\PYTHON\folder1\file.txt
C:/MYPYTHON/Lecture27/file.txt
C:\MYPYTHON\Lecture27\file.txt
C:\MYPYTHON/Lecture27\file.txt
```

Функція не виправляє тип слеша!!!!

Функція **normpath()**

Формат функції: `normpath(<Шлях>)`:

Нехай дано шлях: `'C:/PYTHON/folder1/file.txt'`.

Щоб цей шлях зробити коректним, необхідно скористатися функцією `normpath()`:

Приклад 8.67.

```
import os
p = os.path.join(r"C:\\", "PYTHON/folder1/", "file.txt")
c = os.path.normpath(p)
print("Помилка:",p)
print("Правильно:",c)
```

Результат:

```
Помилка: C:\\PYTHON/folder1/file.txt
Правильно: C:\PYTHON\folder1\file.txt
```

8.3.8. Перенаправлення вводу/виводу

Значення, що повертається методом `fileno()`, завжди буде більшим за число 2:

0 закріплене за стандартним вводом `stdin`,

1 – за стандартним виводом `stdout`,

2 – за стандартним виводом повідомлень про помилки `stderr`.

Усі ці потоки мають деяку подібність із файловими об'єктами. `stdout` і `stderr` підтримують метод `write()`, `stdin` – метод `readline()`. Якщо цим потокам присвоїти посилання на об'єкт, що підтримує файлові методи, то можна перенаправляти стандартні потоки у відповідний файл.

Приклад 8.68.

```
import sys                  # Підключаємо модуль sys
tmp_out = sys.stdout # Зберігаємо посилання на sys.stdout
f = open(r"file.txt", "a") # Відкриваємо файл на дозапис
sys.stdout = f # Перенаправляємо вивід у файл
print("Пишемо рядок1 у файл")
sys.stdout = tmp_out # Відновлюємо стандартний вивід
print("Пишемо рядок в стандартний вивід")
f.close() # Закриваємо файл
f = open(r"file.txt", "r") # Відкриваємо файл на читання
p = f.read()
print(p)
f.close() # Закриваємо файл
```

Результати виконання:

```
Пишемо рядок в стандартний вивід
Пишемо рядок1 в файл
```

Пояснення до попереднього прикладу

1. Зберігаємо у тимчасовій змінній посилання на стандартний вивід `tmp_out`.

tmp_out = sys.stdout

2. Відкриваємо файл на запис: f = open(r"file.txt", "a")

3. Перенаправляємо вивід файлу на sys.stdout: sys.stdout = f

4. Повертаємо вивід в sys.stdout: sys.stdout = tmp_out

5. Закриваємо файл f.close()

6. Відкриваємо файл **file.txt** на читання, зчитуємо дані і бачимо, що ще записані (дозаписані) в нього дані відсутні, оскільки вони були перенаправлені у sys.stdout.

Перенаправлення виводу з використанням функції **print()**

Параметр `file` *функції* `print`

Функція `print()` прямо підтримує перенаправлення виводу.

Для цього використовується параметр **file**, який за замовчуванням посилається на стандартний потік виводу. Наприклад, записати рядок у файл можна так:

Приклад 8.69.

```
>>> f = open(r"file.txt", "a")
>>> print("Пишемо рядок2 у файл", file = f)
>>> f.close()
```

Параметр **flush** *функції* **print**

Параметр **flush** дозволяє виконати безпосереднє збереження даних із проміжного буфера у файл.

Якщо його значення дорівнює `False` (це значення за замовчуванням), збереження буде виконано лише після закриття файлу або після виклику методу `flush()`.

Якщо **flush=True,** то збереження файлу буде після кожного виклику функції `print()`

Приклад 8.70.

```
import io
f = open(r"file.txt", "a+")
print("Пишемо рядок3 у файл", file = f, flush = True)
```

```
print("Пишемо рядок4 у файл", file = f, flush = True)
f.seek(0,io.SEEK_SET)
print(f.read())
f.close()
```

Результат.

Пишемо рядок3 у файл

Пишемо рядок4 у файл

Стандартний ввід **`stdin`**

`sys.stdin` використовується для стандартного інтерактивного вводу даних:

```
>>>import sys
>>>sys.__stdin__
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>
>>>sys.__stdout__
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>
>>>sys.__stderr__¶
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
```

Це об'єкти які містять оригінальні значення для `stdin, stdout` та `stderr` на початку програми.

Стандартний ввід `sys.stdin` та `sys.stderr` також можна перенаправляти.

У цьому випадку функція `input()` буде читати один рядок з файлу при кожному виклику. При досягненні кінця файлу виконується виключення `EOFError`.

Приклад 8.71.

```
import sys
tmp_in = sys.stdin # Зберігаємо посилання на sys.stdin
f = open(r"file.txt", "r")# Відкриваємо файл на читання
sys.stdin = f # Перенаправляємо ввід
while True:
   try:
      line = input()# Читаємо рядок з файлу
      print(line) # Виводимо рядок
```

```
    except EOFError: # Якщо досягнутий кінець файлу,
        break  # виходимо з циклу
sys.stdin = tmp_in # Відновлюємо стандартний ввід
f. close ()# Закрываємо файл
```

Результати виконання:

```
Зчитано рядок1 в файл
Зчитано рядок2 в файл
Зчитано рядок3 в файл
Зчитано рядок4 в файл
```

Метод **isatty()**

Якщо необхідно довідатися, чи посилається стандартний ввід на термінал, можна скористатися методом `isatty()`.

Метод повертає `True`, якщо об'єкт посилається на термінал, і `False` – якщо ні.

Приклад 8.72.

```
import sys,os
if  sys.stdin.isatty():
    print("We use TTY")
f = open(os.path.abspath("C:/MY/file.txt"), "r")
tmp_in = sys.stdin # Зберігаємо посилання на sys.stdin
sys.stdin = f  # Перенаправляємо ввід
if  not sys.stdin.isatty():
    print("We use file", f.name)
sys.stdin = tmp_in # Відновлюємо стандартний ввід
f.close() # Закриваємо файл
```

Командний рядок для перенаправлення вводу/виводу

Перенаправляти стандартне ввід/вивід можна також за допомогою командного рядка. Для прикладу створимо в папці **`C:\PYTHON`** файл **`example26.py`** з кодом, наведеним у прикладі.

Приклад 8.73.

```
import sys
sys.stdout.reconfigure(encoding='cp1251')
while True:
    try:
       line = input ()
       print(line)
    except EOFError:break
```

Результат виконання:

```
ЩО ВВОДИМО, те й виводимо
ЩО ВВОДИМО, те й виводимо
```

1. Відкриваємо термінал Windows у робочому каталозі

2. Тепер виведемо вміст створеного раніше текстового файлу `data.txt` (його вміст може бути будь-яким), виконавши команду:

`python.exe example26.py < data26.txt`

Перенаправляти стандартний вивід у файл можна у аналогічний спосіб. Тільки в цьому випадку символ `< необхідно замінити символом >`.

Запис у файл

Змінимо файл example27.py у такий спосіб:

Приклад 8.74.

```
print ("Вивід перенаправлено у файл")
# Цей рядок буде записаний у файл
```

Тепер перенаправляємо вивід у файл `data27.txt`, виконавши команду:

`python.exe` `example27.py` **`>`** `data27.txt`

У цьому режимі файл `data27.txt` буде перезаписаний.

Результат виконання:

```
file.txt: Вивід перенаправлений у файл
```

8.4. Модулі `pickle` і `shelve`

описування у файл

Якщо необхідно додати результат у кінець файлу, слід використовувати символи **>>**. Приклад дозапису в файл:

Приклад 8.75

Запишемо у файл **example28.py**: `print ("Дані дописані у файл")`

Перейдемо знову в командний рядок

```
python.exe example28.py >> data27.txt
```

Результат виконання:

```
file.txt: Вивід перенаправлений у файл
          Дані  дописані у файл
```

Збереження об'єкту у файл

Зберегти об'єкти у файл і надалі відновити об'єкти з файлу дозволяють модулі `pickle` (засолювати, маринувати) і `shelve` (відкладати в довгу шухляду).

8.4.1. Функція `dump() i load()`:

dump (<Об'єкт>, <Файл>)

Функція виконує серіалізацію об'єкта й записує дані в зазначений файл.

У параметрі <Файл> вказується файловий об'єкт, відкритий на запис у **бінарному режимі**. Приклад збереження об'єкта у файл:

Приклад 8.76.

```
import pickle
f = open(r"file.txt", "wb")
obj = ["Рядок", (2, 3)]
pickle.dump(obj, f)
f.close()
```

Функція **load()**

Функція **load()** читає дані з файлу й перетворює їх в об'єкт. У параметрі <Файл> вказується файловий об'єкт, відкритий на читання в **бінарному режимі**.

Формат функції: load(<Файл>)

Приклад відновлення об'єкта з файлу:

Приклад 8.77.

```
import pickle
#Зчитування об'єкта з файла
f = open(r"file.txt", "rb")
```

```
obj = pickle.load(f)
print(obj)
f.close()
```

Результат:

```
['Рядок', (2, 3)]
```

Збереження кількох об’єктів у файлі

В один файл можна зберегти відразу кілька об'єктів, послідовно викликаючи функцію **dump().** Приклад збереження декількох об'єктів:

Приклад 8.78.

```
import pickle
#Збереження кількох об'єктів у файлі
obj1 = ["Рядок", (2, 3)]
obj2 = (1, 2)
f = open(r"file.txt", "wb")
# Зберігаємо перший об'єкт
pickle.dump(obj1, f)
# Зберігаємо другий об'єкт
pickle.dump(obj2, f)
f. close()
```

Відновлення кількох об’єктів з файлу

Для відновлення об'єктів необхідно кілька раз викликати функцію **load().**

Приклад 8.79.

```
import pickle
f = open(r"file.txt", "rb")
#Відновлюємо перший об'єкт
obj1 = pickle.load(f)
#Відновлюємо другий об'єкт
obj2 = pickle.load(f)
print( obj1, obj2)
f.close()
```

Результат:

```
['Рядок', (2, 3)] (1, 2)
```

```
import pickle
mydata=[]
mydata.append("Дані рядкового типу")
mydata.append(["елемент1","елемент2","3",4])
mydata.append({"a":1,"b":2,"c":3})
mydata.append(b'12345')
mydata.append({11,12,13,"el"})
for i in mydata:
    print(i)
with open(r"complexdat.txt","wb") as f:
    pickle.dump(mydata,f)
```

Результат

```
Дані рядкового типу
['елемент1', 'елемент2', '3', 4]
{'a': 1, 'b': 2, 'c': 3}
b'12345'
{11, 12, 13, 'el'}
import pickle
with open(r"complexdat.txt","rb") as f:
    newdat=pickle.load(f)
for i in newdat:
    print(i)
```

Результат:

```
Дані рядкового типу
['елемент1', 'елемент2', '3', 4]
{'a': 1, 'b': 2, 'c': 3}
b'12345'
{11, 12, 13, 'el'}
```

8.4.2. Класи **Pickler і Unpickler**

Зберегти об'єкт у файл можна також за допомогою **методу dump**(<Об'єкт>) класу **Pickler**. Створення об’єкта має наступний формат:

a=Pickler(<**Файл**>)

Збереження

Приклад збереження об'єкта у файл:

Приклад 8.80

```
import pickle
#Збереження об'єкта методом Pickle.dump
f = open(r"file.txt", "wb")
obj = ["Рядок", (2, 3)]
pkl = pickle.Pickler(f)
pkl.dump(obj)
f.close()
```

Відновлення

Відновити об'єкт із файлу дозволяє метод **load()** із класу **Unpickler**. Створення об’єкта класу **Unpickler**: **b=Unpickler**(<**Файл**>)

Приклад відновлення об'єкта з файлу:

Приклад 8.81

```
import pickle
#Відновлення об'єкта методом load класу Unpickler
f = open(r"file.txt", "rb")
unp = pickle.Unpickler(f)
obj=unp.load()
print(obj)
f.close()
```

Результат:

```
['Рядок', (2, 3)]
```

Перетворення об’єкта у послідовність байтів

Модуль **pickle** дозволяє також перетворити об'єкт у послідовність байтів і відновити об'єкт із послідовності. Для цього призначена функція:

dumps(<Об'єкт>[, <Протокол>])

Функція виконує серіалізацію об'єкта й повертає послідовність байтів спеціального формату.

Формат залежить від зазначеного протоколу –числа від 0 до 4 у порядку від більш старих до більш нових і досконалих.

За замовчуванням використовується протокол 4.

Приклад перетворення списку й кортежу:

Приклад 8.82

```
import pickle
obj1 = [1, 2, 3, 4, 5]  # Список
obj2 = (6, 7, 8, 9, 10) # Кортеж
bin_obj1 = pickle.dumps(obj1)
print(bin_obj1)
bin_obj2 = pickle.dumps(obj2)
print(bin_obj2)
```

Результат:

```
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

Відновлення послідовності байтів у об’єкт

Функція перетворює послідовність байтів спеціального формату в об'єкт.

loads(<Послідовність байтів>)

Приклад відновлення списку й кортежу:

Приклад 8.83.

```
import pickle
obj1 = pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
print(obj1)
obj2 = pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
print(obj2)
```

Результат:

```
[1, 2, 3, 4, 5]
(6, 7, 8, 9, 10)
```

Приклад 8.84.

```
import pickle
obj1 = [1, 2, 3, 4, 5]  # Список
bin_obj1 = pickle.dumps(obj1)
print(obj1)
print(bin_obj1)
f = open(r"file1.txt", "wb")
f.write(bin_obj1)
f.close
f = open(r"file1.txt", "rb")
data = f.read()
f.close
print(data)
print(pickle.loads(data))
```

Результат:

```
[1, 2, 3, 4, 5]
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
[1, 2, 3, 4, 5]
```

8.4.3. Модуль shelve

Модуль **shelve** дозволяє зберігати об'єкти під певним ключем. Ключ задають у вигляді рядка.! Доступ до об’єктів подібний до доступу у словниках.

Відкриття файлу з набором об'єктів виконує функція:

shelve.open().

Функція має наступний формат:

shelve.open(<Шлях до файлу>[,flag="c",protocol=None])

Параметр **flag** задає режими відкриття файлу:

`r` – тільки читання;

`w` – тільки запис;

`c` – читання й запис (значення за замовчуванням). Якщо файл не існує, він буде створений;

`n` – запис. Якщо файл не існує, він буде створений. Якщо файл існує, він буде перезаписаний.

Методи об'єкта **`DbfilenameShelf`**

Функція **`open()`** повертає об'єкт

`shelve.DbfilenameShelf`

За допомогою даного об'єкта проводиться подальша робота з базою даних.

Методи:

`close()` – закриває файл із базою даних.

`keys()` – повертає об'єкт із ключами;

`values()` – повертає об'єкт зі значеннями;

`items()` – повертає об'єкт-ітератор, який на кожній ітерації генерує кортеж, що містить ключ і значення.

Для прикладу створимо файл і збережемо в ньому список і кортеж.

Приклад 8.84. Створення файлу і доступ по ключу

```
import shelve #Підключаємо модуль
db = shelve.open("db1")#Відкриваємо файл
#Зберігаємо список
db["obj1"] = [1, 2, 3, 4, 5]
#Зберігаємо кортеж
db["obj2"] = (6, 7, 8, 9, 10)
# Вивід значень
print(db["obj1"])
print(db["obj2"])
#Закриваємо файл
db.close()
```

Результат:

```
[1, 2, 3, 4, 5]
(6, 7, 8, 9, 10)
```

Застосування методів **shelve.DbfilenameShelf**

Приклад 8.85. Посилання на ітератори

```
import shelve #Підключаємо модуль
db = shelve.open("db1")
print(db.keys())
print(db.values())
print(db.items())
db.close()
```

Результат роботи:

KeysView(<shelve.DbfilenameShelf object at 0x02121910>)

ValuesView(<shelve.DbfilenameShelf object at 0x02121910>)

ItemsView(<shelve.DbfilenameShelf object at 0x02121910>)

```
import shelve #Підключаємо модуль
db = shelve.open("db1")#Відкриваємо файл
db["obj1"] = [1, 2, 3, 4, 5]
db["obj2"] = (6, 7, 8, 9, 10)
print(db["obj1"])
print(db["obj2"])
db.close()
db = shelve.open("db1")
print("Ключі: ")
for i in db.keys():
    print(i, end=",")
print("\nЗначення: ")
for i in db.values():
    print(i, end=",")
print("\nКортежі: ")
for i in db.items():
    print(i, end=",")
db.close()
```

Результат роботи:

```
[1, 2, 3, 4, 5]
(6, 7, 8, 9, 10)

Ключі:
obj1,obj2,

Значення:
[1, 2, 3, 4, 5],(6, 7, 8, 9, 10),

Кортежі:
('obj1', [1, 2, 3, 4, 5]),('obj2', (6, 7, 8, 9, 10)),
```

8.4.4. Метод **get ()**і **setdefault()**

get(<Ключ>[, <Значення за замовчуванням>])

Якщо ключ присутній, то метод повертає значення, відповідне до цього ключа.

Якщо ключ відсутній, то повертається значення **None** або значення, зазначене в другому параметрі.

Приклад 8.86.

```
import shelve    #Підключаємо модуль
db = shelve.open("MyDatabase")#Відкриваємо файл
db["obj1"] = [1, 2, 3, 4, 5]
db["obj2"] = (6, 7, 8, 9, 10)
db["obj3"] = "Рядок"
a=db.get("obj3")
print(a)
a=db.get("obj4", "Неіснуючий рядок")
print(a)
```

Результат:
```
Рядок
Неіснуючий рядок
```

Метод **setdefault**

setdefault(<Ключ>[,<Значення за замовчуванням>])

1.Якщо ключ присутній, то метод повертає значення, відповідне до цього ключа.

2.Якщо ключ відсутній, створюється новий елемент зі значенням, зазначеним у другому параметрі, і як результат повертається це значення.

3.Якщо другий параметр не зазначений, значенням нового елемента буде `None`.

Приклад 8.87.

```
import shelve #Підключаємо модуль
db = shelve.open("MyDatabase")#Відкриваємо файл
db["obj1"] = (9, 14)
db["obj4"]={"c":1,"k":2}
d=(1,2,3)
a=db.setdefault("obj1")
b=db.setdefault("obj4",d)
db.close()
print(a, b)
```

Результат:

```
(9, 14) {'k': 2, 'c': 1}
```

8.4.5. Метод `pop()`, `popitem()` і `setdefault()`

pop(<**Ключ**>[, <**Значення за замовчуванням**>])

1.Видаляє елемент по ключу і повертає його значення.

2.Якщо ключ відсутній, повертається другий параметр.

3.Якщо ключ відсутній, і другий параметр не зазначений, то виконується виключення `Keyerror`.

Приклад 8.89. Видалення методом **pop()**

```
import shelve #Підключаємо модуль
db = shelve.open("MyDatabase")#Відкриваємо файл
db["obj1"] = (9, 14)
d={"c":1,"k":2}
a=db.pop("obj1")
b=db.pop("obj4",d)
# c=db.pop("obj5", "Ключа obj5 немає")
try:    m = db.pop("obj5")
```

```
except KeyError:    print("Такого ключа немає")
db.close()
print(a, b)
```

Результат:

```
Такого ключа немає
(9, 14) {'c': 1, 'k': 2}
```

Приклад 8.90.

```
import shelve #Підключаємо модуль
d={"c":1,"k":2}
db = shelve.open("MyDatabase")#Відкр. Файл
db["obj1"] = (9, 14)
db["obj2"] = (15, 17)
db["obj3"] = "string"
a=db.pop("obj1")
b=db.pop("obj4",d)
m=list(db.items())
db.close()
print(a, b)
print(m)
```

Результат:

```
(9, 14) {'k': 2, 'c': 1}
[('obj3', 'string'), ('obj2', (15, 17))]
```

Метод **popitem()**

popitem() – видаляє довільний елемент і повертає кортеж з ключа й значення. Якщо файл порожній, виконується виключення Keyerror.

Приклад 8.91.

```
import shelve
db = shelve.open("db1")
db["obj1"] = (9, 14)
db["obj2"] = (15, 17)
db["obj3"] = "string"
```

```
d={"c":1,"k":2}
a=db.popitem(); b=db.popitem()
m=dict(db.items())
db.close()
print(a, b);print(m)
```

Результат:

```
('obj1', (9, 14)) ('obj2', (15, 17))
{'obj3': 'string'}
```

Метод **clear()**

clear() – видаляє всі елементи.

Метод нічого не повертає як значення.

Приклад 8.92.

```
import shelve
db = shelve.open("MyDatabase")
db["obj1"] = (9, 14)
db["obj2"] = (15, 17)
db["obj3"] = "string"
db.clear()
m=dict(db.items())
db.close()
print(m)
```

Результат:

```
{}
```

8.4.6. Метод update()

update() – додає елементи. Метод змінює поточний об'єкт і нічого не повертає. Якщо елемент із зазначеним ключем уже присутній, то його значення буде перезаписано.

Формати методу:

update(<Ключ1>=<Значення1>[, ... , <Ключn>=<Значенняn>])

update(<Словник>)

update(<Список кортежів із двома елементами>)

update(<Список списків із двома елементами>)

Працюють також:

1.Функція **len() -** для одержання кількості елементів.

2.Оператор **del** для видалення певного елемента,

3.Оператори **in** і **not in** для перевірки існування або неіснування ключа.

Приклад 8.93.

```
import shelve
#Застосування методу update() у різних варіантах
db = shelve.open("MyDatabase")
db.update(obj1=(9, 14),obj2=(15, 17))
db.update({"obj3":4 ,"obj8":4})
db.update([("obj4",4),("obj5",5)])
db.update([["obj7",7],["obj6",6]])
m=dict(db.items())
db.close()
print(m)
```

Результат:

```
{'obj1': (9, 14), 'obj2': (15, 17), 'obj3': 4, 'obj8': 4,
'obj4': 4, 'obj5': 5, 'obj7': 7, 'obj6': 6}
```

Приклад 8.94. Застосування **len(), in, not in**

```
import shelve
#Перевірка наявності та видалення
db = shelve.open("MyDatabase")
print("{0:25} {1}".format("Розмір бази:",len(db)))
a="obj1" in db
print("{0:25} {1}".format("Наявність об'єкта obj1:",a))
del db["obj1"] # Видалення елемента
a="obj1" in db
print("{0:25} {1}".format("Наявність об'єкта obj1:",a))
b="obj1" not in db
```

```
print("{0:25} {1}".format("Відсутність об'єкта:",b))
db.close()
```

Результат:

```
Розмір бази:                8
Наявність об'єкта obj1:     True
Наявність об'єкта obj1:     False
Відсутність об'єкта:        True
```

8.5. Функції для роботи з каталогами з модуля os

8.5.1. Функції `getcwd(), chdir(),rmdir(), listdir()`

`getcwd()` –повертає поточний робочий каталог.

(get current work directory)

Від значення, що повертається цією функцією, залежить перетворення відносного шляху в абсолютний.

Крім того, важливо пам'ятати, що поточним робочим каталогом буде каталог, з якого запускається файл, а не каталог з файлом, що виконується.

Приклад 8.95.

```
import os
wd=os.getcwd()# Поточний робочий каталог
print(wd)
```

Результат:

```
C:\PYTHON
```

`chdir (<Ім'я каталогу>)` – робить зазначений каталог поточним:

Приклад 8.96.

```
import os
# chdir() робить зазначений каталог поточним
print("{0:<30} {1}".format("Поточний робочий каталог:",os.getcwd()))
a = os.path.abspath("note28.ipynb")
print("{0:<30} {1}".format("Абсолютний шлях:",a))
os.chdir(os.path.abspath("folder1"))
print("{0:<30} {1}".format("Змінили робочий каталог:",os.getcwd()))
```

```
os.chdir(os.path.dirname(a))
print("{0:<30} {1}".format("Повернули робочий каталог:",os.getcwd()))
```

Результат:

```
Поточний робочий каталог:     C:\PYTHON
Абсолютний шлях:              C:\PYTHON\note28.ipynb
Змінили робочий каталог:      C:\PYTHON\folder1
Повернули робочий каталог:    C:\PYTHON
```

`mkdir(<Ім'я каталогу>[, <Права доступу>])` – створює новий каталог із правами доступу, зазначеними в другому параметрі.

Права доступу задають вісімковим числом (значення за замовчуванням 0o777).

Приклад створення нового каталогу в поточному робочому каталозі:

Приклад 8.97

```
import os
try:
    os.mkdir("newfolder")#Створення каталогу
    os.mkdir("newfolder\\subfolder")
    print("Каталог створено")
except FileExistsError:
    print("Каталог уже існує")
```

`rmdir (<Ім'я каталогу>)` – видаляє порожній каталог. Якщо в каталозі є файли або зазначений каталог не існує, виконується виключення – підклас класу `OSError`.

Вилучимо каталог `newfolder`:

Приклад 8.98.

```
import os
# rmdir видаляє пусті існуючі каталоги
try:
    os.rmdir("newfolder\\subfolder") # Видалення
    print("Видалили пустий каталог newfolder\\subfolder")
    os.rmdir("newfolder") # Видалення каталогу
    print("Видалили пустий каталог newfolder")
except OSError:
```

```
    print("Каталог не існує або він не пустий  ")
```

`listdir(<Шлях>)` – повертає список об'єктів у зазначеному каталозі:

Приклад 8.98.

```
import os
#listdir створює список об'єктів, розміщених у каталозі
dl = os.listdir(os.path.dirname(os.path.abspath("note28.ipynb")))
for i in dl:
  print(i)
```

Результат

```
.hypothesis
.idea
data.txt
file.txt
folder1
loadfile.py
main.py
mypack
note28.ipynb
__pycache__
```

8.5.2. Функції `walk(),rmtree(), normcase()`

`walk()` – дозволяє обійти дерево каталогів.

Формат функції:

```
walk(<Початковий каталог>[, topdown=True])
```

`walk()` повертає об'єкт

На кожній ітерації через цей об'єкт доступний кортеж із трьох елементів:

1 елемент-поточний каталог,

2 елемент-список каталогів,

3 елемент-список файлів, що містяться у каталозі.

Параметр `topdown` задає послідовність обходу каталогу.

`Topdown=True` - значення за замовчуванням. У цьому випадку пошук зверху вниз.

Приклад 8.99.

```
import os
for (p, d, f) in os.walk(os.path.dirname(os.path.abspath("note28.ipynb"))):
   print("КАТАЛОГ")
   print(p)
   print("Список підкаталогів")
      print(d)
   print("Список файлів у поточному каталозі")
   print(f,"\n")
```

Результати:

```
КАТАЛОГ
C:\PYTHON
Список підкаталогів
['.idea',  'folder1', 'mypack', '__pycache__']
Список файлів у поточному каталозі
['binfile', 'chng.py', 'data.txt', 'file.txt', 'file1.txt', 'first.py', 'first1.py', 'loadfile.py', 'main.py',
'note28.ipynb', 'proba.txt', 'second.py']

КАТАЛОГ
C:\PYTHON\.idea
Список підкаталогів
['inspectionProfiles']
Список файлів у поточному каталозі
['.gitignore', 'misc.xml', 'modules.xml', 'PYTHON.iml',
'workspace.xml']
КАТАЛОГ
C:\PYTHON\folder1
Список підкаталогів
['folder2']
Список файлів у поточному каталозі
```

```
['f1.txt']

КАТАЛОГ
C:\PYTHON\folder1\folder2
Список підкаталогів
['vkladena']
Список файлів у поточному каталозі
['f2.txt']
```

`Topdown = False` послідовність обходу знизу вверх:

Приклад 8.100.

```
import os
for (p, d, f) in os.walk(os.path.dirname(os.path.abspath("note28.ipynb")), False):
   print("КАТАЛОГ")
   print(p)
   print("Список підкаталогів")
   print(d)
   print("Список файлів у поточному каталозі")
   print(f,"\n")
```

Результати:

```
КАТАЛОГ
C:\PYTHON\folder1\folder2
Список підкаталогів
КАТАЛОГ
C:\PYTHON\folder1
Список підкаталогів
['folder2']
Список файлів у поточному каталозі
['f1.txt']
КАТАЛОГ
C:\PYTHON
```

```
Список підкаталогів
['.idea', '.ipynb_checkpoints', 'folder1', 'mypack', 'vkladena', '__pycache__']
Список файлів у поточному каталозі
['binfile', 'chng.py', 'data.txt', 'file.txt', 'file1.txt', 'first.py', 'first1.py', 'loadfile.py', 'main.py',
'note28.ipynb', 'proba.txt', 'second.py']
```

Завдяки такій послідовності обходу каталогів можна вилучити всі вкладені файли й каталоги. Це особливо важливо при видаленні каталогу, тому що функція **`rmdir()`** дозволяє вилучити тільки порожній каталог.

Приклад очищення дерева каталогів:

Приклад 8.101.

```
import os
p=os.path.dirname(os.path.abspath("note28.ipynb"))
for (p, d, f) in os.walk(p, topdown=False):
    if p.find("folder1")>0:
        print("Поточний каталог:",p)
        for file_name in f:  # Видаляємо всі файли
            print("Знайдено файл:", file_name)
            a = input('Для видалення натисніть "Y":')
            rempath=os.path.join(p, file_name)
            if a == "Y":
                os.remove(rempath)
            if not os.path.exists(rempath):
                    print("Файл успішно видалено")
            else:
                print("Файл", rempath, "не видалено")
```

*УВАГА! Якщо як перший параметр у функції walk() указати кореневий каталог диска, то **всі** наявні в ньому файли й каталоги **будуть вилучені**.*

Функція **`rmtree()`** **з** `shutil`

Вилучити дерево каталогів дозволяє також функція rmtree() з модуля shutil.

Функція має наступний формат:

`rmtree``(<Path>[,<ignore_errors=`**`True`**`>] )`

Якщо < ignore_errors > зазначаємо **False** , то помилка буде оброблятися Функція видаляє вказаний каталог, а також видаляються автоматично:

- всі підкаталоги даного каталогу;
- всі файли у каталозі та підкаталогах.

Приклад 8.102. Видалення непустого каталогу.

```
import shutil
try:
 s = "C:\\PYTHON\\rmfolder"
 shutil.rmtree(s, False)
except FileNotFoundError:
    print("Каталог ",s,"не знайдено")
```

Результат:

```
Каталог C:\PYTHON\rmfolder не знайдено
```

```
import shutil
s = "C:\\PYTHON\\folder0\\"
shutil.rmtree(s, True)
```

Результат:

```
Не буде ніякого виводу
```

normcase (<каталог>) – перетворює заданий шлях до каталогу до вигляду, придатному для використання в поточній операційній системі. В Windows перетворює усі прямі слеші у зворотні. Також у всіх системах приводить усі букви шляху до нижнього регістру.

Приклад 8.103.

```
from os.path import normcase
pt = normcase(r"c:/MYPYTHON/Lec28/file.txt")
print(pt)
pt = normcase(r"c:\MYPYTHON/Lec28/file.txt")
print(pt)
```

```
pt = normcase("c:\MYPYTHON/Lec28/file.txt")
print(pt)
```

Результат:

```
c:\mypython\lec28\file.txt
c:\mypython\lec28\file.txt
c:\mypython\lec28\file.txt
```

Порівняйте з роботою функції **normpath()**

```
from os.path import normpath
pt = normpath(r"c:/MYPYTHON/Lec28/file.txt")
print(pt)
pt = normpath(r"c:\MYPYTHON/Lec28/file.txt")
print(pt)
pt = normpath("c:\MYPYTHON/Lec28/file.txt")
print(pt)
```

Результат

```
c:\MYPYTHON\Lec28\file.txt
c:\MYPYTHON\Lec28\file.txt
c:\MYPYTHON\Lec28\file.txt
```

8.5.3. Функції **isdir(),isfile(), glob()**

`isdir(<Об'єкт>)` – повертає `True`, якщо об'єкт є каталогом, і `False` – якщо ні:

Приклад 8.104.

```
import os.path
if not os.path.isdir(r"C:\PYTHON\file.txt"):
    print("Це не каталог")
if os.path.isdir("C:\\PYTHON"):
    print(r"Це каталог")
if os.path.isdir("C:\\PYTHON\\"):
    print(r"Це також каталог")
```

Результат:

```
Це не каталог
Це каталог
```

Це також каталог

isfile(<Объект>) – повертає True, якщо об'єкт є файлом, і False – якщо ні:

Приклад 8.105.

```
import os.path
if os.path.isfile(r"C:\PYTHON\file.txt"):
    print("Це шлях до файлу file.txt")
if  not os.path.isfile("C:\\PYTHON"):
    print(r"Це не файл")
if  not os.path.isfile("C:\\PYTHON\\"):
    print(r"Це такж не файл")
```

Результат:

```
Це шлях до файлу file.txt
Це не файл
Це такж не файл
```

glob (<Шлях>) повертає шляхи до певної групи файлів чи каталогів

Список шляхів формується з використанням правил регулярних виразів. Функція дозволяє вказати шаблон шляху з такими спеціальними символами:

? - будь-який одиночний символ;

* - будь-яка кількість символів;

[<Символи>] – дозволяє вказати символи, які повинні бути на цьому місці в шляху. Можна перелічити символи. Можна вказати діапазон через дефіс.

Як значення функція повертає список шляхів до об'єктів, які співпадають із шаблоном.

Приклад 8.105.

```
import glob
print("Всі файли з розширенням txt")
pa = glob.glob("C:\\PYTHON\\*.txt")
print(pa)
```

Результат:

```
Всі файли з розширенням txt
```

['C:\\PYTHON\\complexdat.txt', 'C:\\PYTHON\\data.txt', 'C:\\PYTHON\\file.txt',
'C:\\PYTHON\\file1.txt', 'C:\\PYTHON\\proba.txt']

```
import glob
print("Всі файли з розширенням, що вулючає ру ")
pb = glob.glob("C:\\PYTHON\\*.*py*")
print(pb)
```

Результат:

['C:\\PYTHON\\chng.py', 'C:\\PYTHON\\first.py', 'C:\\PYTHON\\first1.py',
'C:\\PYTHON\\loadfile.py', 'C:\\PYTHON\\main.py', 'C:\\PYTHON\\note28.ipynb',
'C:\\PYTHON\\second.py']

```
import glob
print("Всі файли з розширенням символ+ру+2символи ")
pb = glob.glob("C:\\PYTHON\\*.?py??")
print(pb)
```

Результат: Всі файли з розширенням символ+ру+2символи

```
['C:\\PYTHON\\note28.ipynb']
import glob
print("Всі файли з цифрою 1 наприкінці назви")
pd = glob.glob("C:\\PYTHON\\*1.*")
for i in pd:
    print(i)
```

Результат: Всі файли з цифрою 1 наприкінці назви

C:\PYTHON\db1.bak

C:\PYTHON\db1.dat

C:\PYTHON\db1.dir

C:\PYTHON\file1.txt

C:\PYTHON\first1.py

Застосування функції `glob()` *до папок*

Спеціальні символи можуть бути зазначені не тільки в назві файлу, але й в іменах каталогів у шляху. Це дозволяє переглядати одразу кілька каталогів у пошуках об'єктів, відповідних до шаблону.

Приклад 8.106.

```
import glob
p = glob.glob("C:\\PYTHON\\*\\*.txt")
print(p)
```

Результат:

```
['C:\\PYTHON\\folder1\\file1.txt',
'C:\\PYTHON\\mypack\\file.txt',
'C:\\PYTHON\\vkladena\\data.txt',
'C:\\PYTHON\\vkladena\\file.txt',
'C:\\PYTHON\\vkladena\\file1.txt']
```

Виключення, які виникають при файлових операціях

При вивченні виключень зазначалось, що функції й методи, що здійснюють файлові операції, при виникненні позаштатних ситуацій викликають виключення класу `OSError` або одне з виключень, що є його підкласом.

Виключень-підкласів класу `OSError` багато. Ми розглянемо ті з них, що стосуються саме операцій з файлами й папками:

`BlockingIOError` – не вдалося заблокувати об'єкт (файл або потік вводу/виводу).

`ConnectionError` – помилка мережного з'єднання. Може виникнути при відкритті файлу по мережі. Є базовим класом для ряду інших виключень більш високого рівня, описаних у документації по `Python`.

`FileExistsError` – файл або папка із заданим іменем уже існують.

`FileNotFoundError` – файл або папка із заданим іменем не виявлені;

`InterruptedError` – файлова операція несподівано перервана з якої-небудь причини;

`IsADirectoryError` – замість шляху до файла зазначений шлях до папки;

`NotADirectoryError` – замість шляху до папки зазначений шлях до файла;

`PermissionError` – відсутні права на доступ до зазначеного файлу або папки;

`TimeoutError` – вийшов час, відведений системою на виконання операції.

Приклад коду, що обробляє деякі із зазначених виключень, наведений нижче.

Приклад 8.107.

```
try:
 f=open(r"C:\temp\new\file.txt")
except FileNotFoundError:
    print("Файл відсутній")
except IsADirectoryError:
    print("Це не файл, а папка")
except PermissionError:
    print("Відсутні права на доступ до файла")
except OSError:
    print("Невизначена помилка відкриття файла")
```

Результат: Файл відсутній

Контрольні запитання до розділу 8.

1.Який тип об’єкту повертає функція `open`?

2. Чим відрізняється відносний та абсолютний шлях до файлу?

3.Які параметри функції `open()` найчастіше використовують при доступі до файлів?

4. Перерахуйте відомі відомі вам модифікації режимів читання-запису.

5. Опишіть умови застосування менеджерів контексту `With.. as`

6. Які методи використовують для читання файла?

7. Опишіть методи застосування запису у файл.

9. Які способи перебору послідовностей використовують?

10. Опишіть принципи роботи методу `flush()` та `fileno()`.

11. Наведіть приклад використання методу `seek()`

12. Які варіанти застосування функції копіювання існують?

13. Опишіть застосування функції для переміщення файлів.

14. Які функції вилучення файлів вам відомі?

15. Опишіть принципи застосування прав доступу до файлу.

9. ГРАФІЧНИЙ ІНТЕРФЕЙС КОРИСТУВАЧА. БІБЛІОТЕКА `tkinter`

9.1. Початкові відомості про графічний інтерфейс користувача

У різноманітті програм, які пишуть програмісти, виділяють додатки із графічним інтерфейсом користувача (GUI-graphical user interface).

При створенні таких програм стають важливими не тільки алгоритми обробки даних, але й розробка для користувача програми зручного інтерфейсу, взаємодіючи з яким, він буде визначати поведінку програми.

https://rauterberg.employee.id.tue.nl/lecturenotes/MS-Official-GUI-2001.pdf

«Official Guidelines for User Interface Developers and Designers»

Сучасний користувач в основному взаємодіє з програмою за допомогою різних кнопок, меню, віконець, вводячи інформацію в спеціальні поля, вибираючи певні значення в списках і т. д. Ці "зображення" у певному сенсі й формують GUI, в подальшому ми їх будемо називати ***віджетами*** (від англ. widget "штучка").

9.1.1. Послідовність кроків при створенні GUI

Послідовність кроків при створенні графічного додатка має свої особливості. Програма повинна виконувати своє основне призначення, бути зручною для користувача, реагувати на його дії.

Не вдаючись у подробиці розробки, розглянемо, які етапи приблизно потрібно пройти при програмуванні, щоб одержати програму з GUI:

1. Імпорт бібліотеки
2. Створення головного вікна
3. Створення віджетів
4. Установка їх властивостей
5. Визначення подій
6. Визначення обробників подій
7. Розташування віджетів на головному вікні
8. Відображення головного вікна

9.1.2. Завантаження tkinter та створення головного вікна

Імпорт модуля tkinter

Для Python віджети включені в спеціальну бібліотеку tkinter.

Якщо її імпортувати в програму, то можна користуватися її компонентами, створюючи графічний інтерфейс.

Як і будь-який модуль, tkinter в Python можна імпортувати двома способами:

1. командою **import** tkinter
2. командою **from** tkinter **import** *.

У версії Python 3 ім'я модуля пишеться з малої букви (tkinter), хоча в більш ранніх версіях використовувалася прописна (Tkinter). Отже, перший рядок програми повинен мати такий вигляд:

```
from tkinter import *
```

Створення головного вікна

У сучасних операційних системах будь-який додаток користувача (застосунок) (приложение (рус.), application(eng.)) розташований у вікні, яке можна назвати головним, тому що в ньому розміщуються всі інші віджеты.

Об'єкт вікна верхнього рівня створюється при звертанні до класу **Tk** модуля **tkinter.** Змінну, пов'язану з об'єктом-вікном, прийнято називати **root** (хоча зрозуміло, що можна назвати як завгодно). Другий рядок коду:

```
root = Tk()
```

Відображення головного вікна

Головне вікно з'явиться після виклику спеціального методу mainloop:

```
root.mainloop()
```

Даний рядок коду повинен бути завжди наприкінці скрипта!

Приклад 9.1.

```
from tkinter import *
root = Tk()
root.mainloop()
```

9.1.3. Створення віджетів

Розглянемо встановлення кнопки. Кнопка створюється при звертанні до класу `Button` модуля `tkinter`. Об'єкт кнопка зв'язується з якою-небудь змінною.

У класу `Button` (як і всіх інших класів, за винятком `Tk`) є обов'язковий параметр — об'єкт, якому кнопка належить (кнопка не може "бути нічийною").

Поки в нас є єдине вікно (`root`), воно й буде аргументом, переданим у клас при створенні об'єкта-кнопки: `but = Button(root)`

Розміщення віджетів

В будь-якому додатку віджети не розкидані по вікну аби як, а добре організовані.

Нехай потрібно просто кнопку будь-яким чином відобразити у вікні. Найпростіший спосіб — це використання методу **`pack.`**

```
but.pack()
```

Без наявності даного коду, відображення віджета у полі вікна не відбудеться, оскільки не визначено місце його розташування

Установка властивостей віджетів

У кнопки багато властивостей:

розмір,

колір тла,

написи,

зображення та ін.

Ми розглянемо їх пізніше.

Установимо всього одну властивість — текст напису (text):

```
from tkinter import *
root = Tk()
but=Button(root)
but["text"] = "Друк\nнашого тексту"
but.pack()
root.mainloop()
```

Визначення подій і їх обробників

При натисканні на кнопку повинні виконуватись певні дії. Дії (алгоритм), які відбуваються при тій або іншій події, можуть бути досить складним.

Тому часто їх оформляють у вигляді функції, а потім викликають, коли вона знадобиться.

Припустимо, що задача кнопки вивести яке-небудь повідомлення в потік виводу, використовуючи функцію print.

Тоді друк на екран оформимо у вигляді функції `printer`:

```
def printer(event):
    print ("Як завжди, черговий 'Hello World!'")
```

Параметр `event` – це яка-небудь подія.

Потрібно зв'язати цю подію з обробником (функцією `printer`). Для зв'язку призначений метод `bind`. Синтаксис зв'язування події з обробником має такий вигляд: `but.bind("<Button-1>",printer)`

Код програми має такий вигляд:

Приклад 9.2.

```
from tkinter import *
def printer(event):
    print ("Як завжди, черговий 'Hello World!'")
root = Tk()
but = Button(root)
but["text"] = " Друк\nнашого тексту"
but.bind("<Button-1>",printer)
but.pack()
root.mainloop()
```

Об'єктно-орієнтований підхід

Приклад 9.3.

```
from tkinter import *
class MyClass:
```

```
    def __init__(self):
        self.but = Button(root)
        self.but["text"] = "Друк\nнашого тексту"
        self.but.bind("<Button-1>",self.printer)
        self.but.pack()
    def printer(self,event):
        print ("Як завжди, черговий 'Hello World!'")
root = Tk()
obj = MyClass()
root.mainloop()
```

9.1.4. Огляд віджетів бібліотеки tkinter

Кнопки

Об'єкт-Кнопка створюється викликом класу `Button` модуля `tkinter`.

Обов'язковим аргументом є лише батьківський віджет (наприклад, вікно верхнього рівня).

Інші властивості можуть вказуватися при створенні кнопки або задаватися (змінюватися) пізніше.

Синтаксис:

```
<змінна> = Button (<батьківський віджет>,
[властивість=значення, … …])
```

У кнопки багато властивостей, у прикладі зазначені лише деякі з них.

Кнопка з модифікованими властивостями

Приклад 9.4.

```
from tkinter import *
root = Tk()
but = Button(root,
text="Це наша кнопка", #напис на кнопці
width=30,height=5, #ширина и висота
bg="white",fg="blue") #колір тла і напису
```

```
but.pack()
root.mainloop()
```

Властивості й методи кнопки:

Властивість	Опис
activebackground	Колір тла, коли кнопка перебуває під курсором
activeforeground	Колір переднього плану, коли кнопка перебуває під курсором.
bd	Ширина рамки в пікселях. За замовчуванням дорівнює 2.
bg	Нормальний колір тла.
command	Функція або метод, який буде викликатися при натисканні кнопки.
fg	Нормальний передній план (текст) колір.
font	Шрифт тексту, який буде використовуватися для друку лейбла кнопки.
height	Висота кнопки в текстових рядках (для текстових кнопок) або в пікселях (для зображень).
image	Зображення, яке буде відображатися на кнопці (замість тексту).
justify	Як показати кілька рядків тексту: LEFT- притиснути ліворуч кожний рядок; CENTER- центрувати; RIGHT- притиснути вправо кожний рядок.
padx	Додаткове заповнення ліворуч і праворуч від тексту.
pady	Додаткове заповнення зверху й знизу від тексту.
relief	Рельєф визначає тип границі. Деякі зі змінних SUNKEN (утоплена), RAISED(піднята), GROOVE (з канавкою) і RIDGE(з обводом).
state	Установіть цю опцію в DISABLED, щоб кнопка стала сірою й недоступною. Якщо має значення ACTIVE, те

	стає активною, коли над нею покажчик миші. За замовчуванням NORMAL.
underline	За замовчуванням дорівнює -1, що означає, що символи тексту на кнопці не підкреслені. Якщо додатне, то відповідний текстовий символ буде підкреслений.
width	Ширина кнопки в символах (для відображення тексту) або пікселях (якщо відображення зображення).
Wraplength	Якщо це значення встановлене в додатне число, текстові рядки будуть завернуті, щоб відповідати довжині.

Використані методи для кнопки.

Medthod	Description
config()	Застосовуємо для зміни атрибутів кнопки після створення об'єкта.
flash()	Викликає блискання кнопки кілька разів між активним і неактивним положенням.

Приклад 9.5. Кнопка з флешом

```
from tkinter import *
def blink(event):
    but_fl.config(bg = 'red')
    but_fl.flash()
    but_fl.after(500, lambda: but_fl.config(bg = 'lightgrey'))
root = Tk()
but_fl = Button(root,text="ЦЕ КНОПКА")
but_fl.config(bg = 'yellow')
but_fl.pack()
but_fl.bind("<Button-1>",blink)
root.mainloop()
```

Використання **messagebox**

Приклад 9.6.

```
from tkinter import *
from tkinter import messagebox
def helloCallBack():
   msg=messagebox.showinfo( "Python", "Hello World")
root = Tk()
root.geometry("200x200")
but = Button(root, text ="Hello",
command = helloCallBack,
bd=4, bg="blue", width=10,fg="white",
activeforeground="black" )
but.place(x=75,y=75)
root.mainloop()
```

Лейбли

Лейблы (або написи) — це досить прості віджети, що містять рядок (або кілька рядків) тексту і використовуються в основному для інформування користувача.

Синтаксис:

<змінна> = Label(<батьківський віджет>,<властивість>, ...)

<батьківський віджет**>** представляє об'єкт, який є предком даного віджета.

<властивість**>** – існує список найбільш часто використовуваних властивостей для цього віджета.

Ці параметри можуть бути використані як пари ключ-значення, розділені комами.

Властивість	Опис
anchor	Цей параметр визначає, де текст позиціонується, якщо віджет має більше простору, ніж потрібно для тексту. За замовчуванням anchor=CENTER, який центрує текст у доступному просторі.
bg	Нормальний колір тла відображається за лейблом.

Властивість	Опис
bitmap	Установіть цей параметр рівним *bitmap* або *image* об'єкту, і лейбл відображатиме цю графіку.
bd	Розмір границі навколо індикатору. За замовчуванням 2 пікселя.
cursor	Якщо встановити в цей параметр вид курсору (*arrow, dot* і т.д.), курсор миші зміниться, коли буде перебувати над лейблом.
font	Якщо ви відображаєте текст у цьому лейблі (використовуючи властивість text або textvariable) указуйте тут шрифт, який буде відображатися.
fg	Якщо ви відображаєте текст або bitmap у цьому лейблі, ця властивість визначає колір тексту.
height	Розмір по вертикалі
image	Для відображення статичного зображення необхідно встановити сюди об'єкт image.
justify	Як показати кілька рядків тексту: LEFT- притиснути ліворуч кожний рядок; CENTER- центрувати; RIGHT- притиснути вправо кожний рядок
padx	Більше простору додається ліворуч і праворуч від тексту всередині віджета. Значення за замовчуванням 1.
pady	Додаткове заповнення ліворуч і праворуч від тексту.
relief	Визначає зовнішній вигляд декоративної границі навколо лейбла. Значення за замовчуванням FLAT. **FLAT,RAISED,SUNKEN,GROOVE,RIDGE**

Властивість	Опис
`text`	Щоб відобразити один або кілька рядків тексту у віджеті лейбла, установіть цей параметр у рядок, що містить текст. Внутрішні символи нового рядка ("\n"), розрив рядка.
`width`	Ширина етикетки(лейбла) в символах (не в пікселях!). Якщо цей параметр не заданий, мітка буде мати такий розмір, щоб відповідати його змісту.
`wraplength`	Ви можете обмежити кількість символів у кожному рядку, установивши цей параметр на потрібне число. Значення за замовчуванням 0 означає, що рядки будуть розбиті тільки на нових рядках.

Приклад 9.7. Приклад використання **`Label`**

```
from tkinter import *
root = Tk()
root.title("Приклад лейблів")
root.geometry("500x500")
lab1 = Label(root, text="Це мітка! \n з двох рядків.", font="Arial 18")
lab1.place(x=150,y=100)
lab2 = Label(root, text="Це мітка з околом 10 пікс",
bd=10, fg = "Yellow", bg = "Green", font="Arial 18", cursor = "dot")
lab2.place(x=100,y=200)
photo=PhotoImage(file="python.gif")
lab3 = Label(root, image = photo,width="150",height="100")
lab3.place(x=160,y=300)
root.mainloop()
```

Однорядкове текстове поле `Entry`

Таке поле створюється викликом класу Entry модуля tkinter. В нього користувач може ввести тільки один рядок тексту.

```
ent = Entry(root,width=20,bd=3)
```

`bd` – це скорочення від borderwidth (ширина границі).

`width` - задає довжину елемента в знакомісцях.

`s = ent.get()` – зчитує введений текст.

Приклад 9.8. Приклад використання Entry

```
from tkinter import *
top = Tk()
top.geometry("500x500")
lb=Label(text= "Введіть дані",font = ("Arial",20))
lb.place(x=140,y=150)
en =Entry(top,width=20,bd=3, font = ("Arial",20))
en.place(x=90,y=200)
def callback():
   print(en.get())
b = Button(top, text="get", width=10, command=callback, font = ("Arial",20))
b.place(x=150,y=250)
top.mainloop()
```

Багаторядкове текстове поле

Text призначений для надання користувачеві можливості введення не одного рядка тексту, а суттєво більше.

```
tex = Text(root,width=40,font="Verdana 12",wrap=WORD)
```

Остання властивість (wrap) залежно від свого значення дозволяє переносити текст, що вводиться користувачем, або по символах (wrap=CHAR), або по словах (wrap=WORD), або взагалі не переносити (wrap=NONE), поки користувач не натисне `Enter`.

Методи для редагування текстової інформації з Text

для зчитування get(),

для вставки- insert(),

для видалення - del()

Метод **insert -** вставка тексту в задану позицію.

Формат: **insert(index, text)**

Для параметра **index** зазвичай вибирають:

INSERT

END

CURRENT

Якщо у першому параметрі задати **INSERT**, то текст, який задано у другому параметрі буде вставлений у поточну позицію курсора.

При задаванні **END** текст вставляють у кінці тексту.

CURRENT – символ, найближчий до вказівника миші.

Приклад 9.9. Застосування методу `insert`

```
from tkinter import *
root=Tk()
lb=Label(root,text="Введіть дані",font=("Arial",20))
en =Entry(root,width=20,bd=3,font=("Arial",20))
text1=Text(root,height=3,width=40,font=("Arial",
14), wrap=WORD)
def callback():
   text1.insert(INSERT, en.get())
b = Button(root, text="Додати", width=10, command=callback, font = ("Arial",20))
lb.pack()
en.pack()
b.pack()
text1.pack()
root.mainloop()
```

Метод **delete.**

Видаляє символ (або вбудований об'єкт) з заданої позиції.

Формат: **delete(start, end=None)**

Перший аргумент: start

1. delete(INSERT) – видаляємо один символ після курсора
2. delete(INSERT, END)- видаляємо всі символи від курсора до кінця
3. delete(1.0, 1.3) - видаляємо три перші символи першого рядка
4. delete(1.0, 2.0) – видаляємо перший рядок

Формат позиції: **x.y**, де **x** – це рядок, що змінюється від 1, а **y** – стовпець (від 0).

Приклад 9.10. Застосування методу **delete**

```
from tkinter import *
def callback():
  text1.delete(1.0, 1.3   )
root=Tk()
lb=Label(root,text= "Введіть дані",font = ("Arial",20))
text1=Text(root,height=3,width=40, font=("Arial",14),wrap=WORD)
b = Button(root, text="Очистити", width=10, command=callback, font = ("Arial",20))
lb.pack()
text1.pack()
b.pack()
root.mainloop()
```

Метод **get.**

Зчитує текст, починаючи з позиції **start** і до позиції **stop.**

Формат: **get(start, stop)**

Варіанти виконання зчитування:

1. `get(1.0, 1.5)` – зчитуємо 5 перших символів (0,1,2,3,4)
2. `get(1.0, 2.0)` – зчитуємо перший рядок
3. `get(1.0, END)` – зчитуємо всі символи
4. `get (INSERT,END)` зчитуємо від курсора до кінця тексту

Приклад 9.11. Застосування методу **`get`**

```
from tkinter import *
def callback():
    lb.config(text = text1.get(start,stop))
def callback2():
    lb.config(text = text1.get(INSERT,END))
root=Tk()
start=1.0
stop=2.0
lb=Label(root,text= "Зчитані дані",font = ("Arial",20))
text1=Text(root,height=3,width=40,
font=("Arial",14),wrap=WORD)
b = Button(root, text="Зчитати перший рядок", width=20,
command=callback, font = ("Arial",20))
b2 = Button(root, text="Читаємо від курсора ", width=20,
command=callback2, font = ("Arial",20))
lb.pack()
text1.pack()
b.pack()
b2.pack()
root.mainloop()
```

Віджет **`Listbox`**

`Listbox` – це віджет, який являє собою список, з елементів якого користувач може вибирати один або кілька пунктів. Має додаткову властивість **`selectmode`**,

Значення: (`BROWSE,SINGLE, MULTIPLE,EXTENDED`)

`selectmode = SINGLE`, дозволяє користувачеві вибрати тільки один елемент списку

`selectmode = EXTENDED`, дозволяє користувачеві вибрати будь-яку кількість елементів списку

`insert(index, *elements)-` вставка елементів

`delete(start,stop) –` видалення елементів

`get(first, last=None)-` зчитування елементів

`curselection()` - повертає набір індексів виділених елементів

`size`() – повертає кількість елементів

Приклад 9.12.

```
from tkinter import *
root=Tk()
root.title("СПИСОК")
en =Entry(root, width=20,bd=3, font = ("Arial",20))
en.insert(END,"КПІ")
lab_ext=Label(text= "EXTENDED",font = ("Arial",14))
lbox_ext=Listbox(root,height=5,width=15, font=("Arial",16))
lab_singl=Label(text= "SINGLE",font = ("Arial",14))
lbox_singl=Listbox(root,height=5,width=15,
font=("Arial",16))
list1=["Київ","Хмельницький","Львів","Одеса"]
list2=["Канберра","Сідней","Мельбурн","Аделаїда"]
for i in list1:
    lbox_ext.insert(END,i)
for i in list2:
    lbox_singl.insert(END,i)
def callback():
    lbox_ext.insert(ACTIVE,en.get())
b = Button(root, text="Insert", width=10, command=callback,
```

```
font = ("Arial",20))
def callback1():
    lbox_singl.delete(ACTIVE,END)
b1 = Button(root, text="Delete", width=10,
command=callback1, font = ("Arial",20))
def callback2():
    lab_ext.config(text=lbox_ext.get(ACTIVE))
b2 = Button(root, text="get", width=10, command=callback2,
font = ("Arial",20))
def callback3():
    lbox_singl.insert(ACTIVE,lbox_ext.get(ACTIVE))
b3 = Button(root, text="Move", width=10, command=callback3,
font = ("Arial",20))
en.pack()
b.pack()
lab_ext.pack()
lbox_ext.pack()
lab_singl.pack()
lbox_singl.pack()
b1.pack()
b2.pack()
b3.pack()
root.mainloop()
```

Віджет **Frame**

Віджет Frame (рамка) призначений для організації віджетів усередині вікна.

Приклад 9.13.

```
# encoding: utf-8
from tkinter import *
root=Tk()
frame1=Frame(root,bg='green',bd=5)
```

```
frame2=Frame(root,bg='red',bd=5)
button1=Button(frame1,text='Перша кнопка')
button2=Button(frame2,text='Друга кнопка')
frame1.pack()
frame2.pack()
button1.pack()
button2.pack()
root.mainloop()
```

Властивість `bd` відповідає за товщину краю рамки.

Віджет **Checkbutton**

`Checkbutton` – дозволяє відзначити „галочкою“ пункт у вікні. При використанні декількох пунктів потрібно кожному присвоїти свою змінну.

`IntVar()` – спеціальний клас бібліотеки для роботи із цілими числами.

`StringVar()` – спеціальний клас бібліотеки для роботи із рядками.

`variable` - властивість, відповідальна за прикріплення до віджету змінної.

`onvalue, offvalue` – властивості, які присвоюють прикріпленій до віджету змінній значення, яке залежить від стану (`onvalue` – при вибраному пункті, `offvalue` – при невибраному пункті).

Приклад 9.14.

```
from tkinter import *
root=Tk()
var1=IntVar()
var2=StringVar()

def fch1():
    print(var1.get())
def fch2():
    print(var2.get())
check1=Checkbutton(root,text='1 пункт для вибору',
```

```
variable=var1,onvalue=10,offvalue=0,command=fch1,
font="Arial 16 bold")
check2=Checkbutton(root,text='2 пункт для вибору',
variable=var2, onvalue="Yes", offvalue="No", command = fch2,
font="Arial 16 bold")
check1.pack()
check2.pack()
root.mainloop()
```

Віджет **Radiobutton**

Віджет Radiobutton виконує функцію, схожу з функцією віджета Checkbutton. Різниця в тому, що у віджеті Radiobutton користувач може вибрати лише один з пунктів. Реалізація цього віджета дещо інша, ніж віджета Checkbutton:

variable - властивість, яка закріплює змінну до Radiobutton.

value – властивість, яка надає значення прикріпленій до віджету змінній.

anchor=W – встановлює положення якоря «захід - West»

Дані встановлюємо та зчитуємо через методи змінної get() та set().

Приклад 9.15.

```
from tkinter import *
root=Tk()
var=IntVar()
vars=StringVar()
vars.set("one")
var.set(1)
def func1():
    print(var.get())
def func2():
    print(vars.get())
rbutton1=  Radiobutton(root, text='1',      variable=var,
```

```
value=1)
rbutton2=  Radiobutton(root, text='2',      variable=var,
value=2)
rbutton3=  Radiobutton(root, text='3',      variable=var,
value=3)
rbutton1s=Radiobutton(root,text='one',   variable=vars,
value="one")
rbutton2s=Radiobutton(root,text='two',   variable=vars,
value="two")
rbutton3s=Radiobutton(root,text='three',variable=vars,
value="three")
button1=Button(root,text='Чтаємо першу групу',
command=func1)
button2=Button(root,text='Чтаємо другу групу',
command=func2)
rbutton1.pack(anchor=W)
rbutton2.pack(anchor=W)
rbutton3.pack(anchor=W)
rbutton1s.pack(anchor=W)
rbutton2s.pack(anchor=W)
rbutton3s.pack(anchor=W)
button1.pack(anchor=W)
button2.pack(anchor=W)
root.mainloop()
```

Приклад 9.16.

```
from tkinter import *
root = Tk()
v = IntVar()
v.set(1)
languages = [("Python",1), ("Perl",2), ("Java",3),
```

```
("C++",4), ("C",5)]
lb = Label(root, text="""Choose your favourite programming
language:""",
justify = LEFT, padx = 20)
lb.pack()
def ShowChoice():
    print(v.get())
for txt, val in languages:
    rbl=Radiobutton(root, text=txt, padx = 20, variable=v,
command=ShowChoice, value=val)
    rbl.pack(anchor=W)
mainloop()
```

Віджет **Scale**

`Scale` (шкала) – це віджет, що дозволяє вибрати будь-яке значення із заданого діапазону.

Властивості:

- **orient** – як розташована шкала у вікні. Можливі значення: `HORIZONTAL`, `VERTICAL` (горизонтально, вертикально).
- **length** – довжина шкали.
- **from_** – з якого значення починається шкала.
- **to** – яким значенням закінчується шкала.
- **tickinterval** – інтервал, через який відображаються мітки шкали.
- **resolution** – крок пересування (мінімальна довжина, на яку можна пересунути движок)

Приклад 9.17

```
from tkinter import *
root = Tk()
def getV(root):
  a = scale1.get()
```

```
   print("Значення", a)
scale1 = Scale(root,orient=HORIZONTAL,length=300, from_=50,
to=80,tickinterval=5,resolution=1)
button1 = Button(root,text="Отримати значення")
scale1.pack()
button1.pack()
button1.bind("<Button-1>",getV)
root.mainloop()
```

Тут використовується спеціальний метод `get()`, який дозволяє зняти з віджета певне значення, і використовується не тільки в `Scale`.

Scrollbar

Цей віджет дає можливість користувачеві "прокрутити" інший віджет (наприклад, `listbox`) і часто буває корисним. Використання цього віджета досить нетривіально. Необхідно зробити дві прив'язки:

command смуги прокручування прив'язуємо до методу **xview/yview** віджета,

а **xscrollcommand/yscrollcommand** віджета прив'язуємо до методу **set** смуги прокручування.

Приклад 9.18.

```
from tkinter import *
root = Tk()
root.title("Скролбар")
scrollbar = Scrollbar(root)
scrollbar.pack(side=RIGHT, fill=Y)
listbox = Listbox(root, font="Arial 16 bold")
listbox.pack()
for i in range(100):
    listbox.insert(END, i)
# attach listbox to scrollbar
listbox.config(yscrollcommand=scrollbar.set)
```

```
scrollbar.config(command=listbox.yview)
mainloop()
```

9.1.5. Огляд пакувальників

Пакувальник або менеджер геометрії, менеджер розташування.

Пакувальник - це спеціальний механізм, який розміщає (упаковує) віджети на вікні.

В `tkinter` є три пакувальники:

`pack, place, grid.`

Зверніть увагу, що в одному віджеті можна використовувати тільки один тип упакування, при змішуванні різних типів упакування програма, швидше за все, не буде працювати. Розглянемо кожний з них.

Пакувальник **`pack()`**

Менеджер геометрії **`pack()`** пакує віджети в рядки або стовпці.

Для керування цим менеджером можна використовувати його режим за замовчуванням, або задавати такі опції геометрії:

`fill`, **`expand`** і **`side`**.

Застосування за замовчуванням

Розміщуємо віджет усередині контейнера-віджета, і він заповнює весь батьківський віджет.

Приклад: `Listbox` у кореневому вікні:

Приклад 9.19. Повне заповнення батьківського віджета

```
from tkinter import *
root = Tk()
root.title("Packe")
listbox = Listbox(root, font="Arial 16 bold")
listbox.pack()
for i in range(20):
    listbox.insert(END, str(i))
mainloop()
```

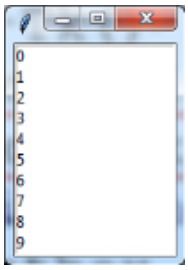


Рис. 9.1. Заповнення вікна віджетом Listbox

Пакування без розтягування

За замовчуванням `Listbox` має розмір 10 елементів. Однак наш `Listbox` містить 20 елементів. Але якщо спробувати показати їх усі за допомогою зміни розмірів вікна, то `tkinter` додасть відступи навколо `Listbox`:

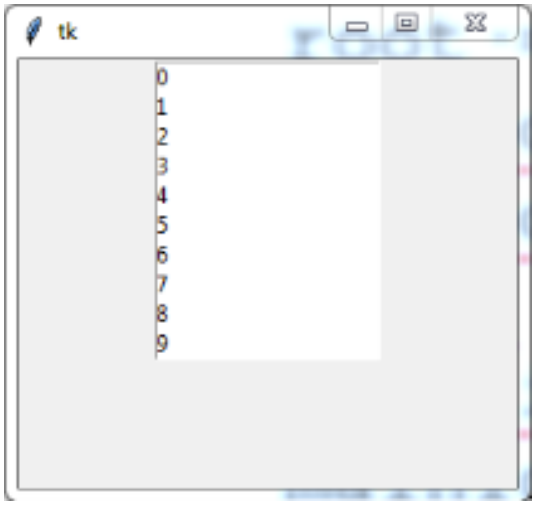


Рис. 9.2. Модифікація вікна без модифікації розміру віджету `Listbox`

Приклад 9.20. Пакування з розтягуванням

```
from tkinter import *
root = Tk()
root.title("Гумова верстка")
listbox = Listbox(root, font= "Arial 16 bold")
for i in range(20):
    listbox.insert(END, str(i))
listbox.pack(fill=BOTH, expand=True)
mainloop()
```

Опція **`fill`** указує менеджерові, що віджет прагне заповнити весь простір, призначений для нього.

Опція має допустимі значення:

1) `fill=BOTH` - віджет розширюється як горизонтально, так і вертикально.

2) `fill=X` - віджет розширюється тільки горизонтально.

3) `fill=Y` - віджет розширюється тільки вертикально.

4) `expand=True`- опція дозволяє розширювати віджет.

Упакування віджетів один над одним

Щоб помістити кілька віджетів у стовпці, можна використовувати метод **`pack()`** без будь-яких опцій:

Приклад 9.21. Упакування віджетів один над одним.

```
from tkinter import *
root = Tk()
root.title("Простий pack")
w = Label(root, text="Червона label",bd=20, bg="red",
fg="white", font="Arial 16 bold")
w.pack()
w = Label(root, text="Зелена label",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack()
w = Label(root, text="Синя label", bd=20, bg="blue",
fg="white", font="Arial 16 bold")
w.pack()
mainloop()
```

Упакування віджетів з горизонтальним розширенням

Приклад 9.22. Розтягування віджетів по ширині батьківського віджета.

```
from tkinter import *
root = Tk()
root.title("Гума по X")
w = Label(root, text="Червона label",bd=20, bg="red",
```

```
fg="white", font="Arial 16 bold")
w.pack(fill=X, expand=True)
w = Label(root, text="Зелена label",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack(fill=X, expand=True)
w = Label(root, text="Синя label", bd=20, bg="blue",
fg="white", font="Arial 16 bold")
w.pack(fill=X, expand=True)
mainloop()
```

Упакування віджетів з вертикальним розширенням

Приклад 9.23. Розтягування віджетів по висоті батьківського віджета.

```
from tkinter import *
root = Tk()
root.title("Гума по Y")
w = Label(root, text="Червона label",bd=20, bg="red",
fg="white", font="Arial 16 bold")
w.pack(fill=Y, expand=True)
w = Label(root, text="Зелена label",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack(fill=Y, expand=True)
w = Label(root, text="Синя label", bd=20, bg="blue",
fg="white", font="Arial 16 bold")
w.pack(fill=Y, expand=True)
mainloop()
```

Упакування віджетів одного поруч із іншим

Для упакування віджетів поруч, використовуємо опцію **side.**

Приклад 9.24.

```
from tkinter import *
root = Tk()
root.title("Віджети поряд")
```

```
w = Label(root, text="Червона",bd=20, bg="red",
fg="white",font="Arial 16 bold")
w.pack(side=LEFT)
w = Label(root, text="Зелена",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack(side=LEFT)
w = Label(root, text="Синя", bd=20, bg="blue", fg="white",
font="Arial 16 bold")
w.pack(side=LEFT)
mainloop()
```

Упакування віджетів з вертикальним розширенням

Якщо необхідно зробити віджети по висоті рівними батьківському, використовуємо додатково опцію `fill=Y`.

Приклад 9.25.

```
from tkinter import *
root = Tk()
root.title("Віджети поряд і гума по Y")
w = Label(root, text="Червона",bd=20, bg="red", fg="white",
font="Arial 16 bold")
w.pack(side=LEFT,fill="y", expand=True)
w = Label(root, text="Зелена",bd=20, bg="green", fg="black",
font="Arial 16 bold")
w.pack(side=LEFT,fill=Y,  expand=True)
w = Label(root, text="Синя",bd=20, bg="blue",
fg="white",font="Arial 16 bold")
w.pack(side=LEFT,fill=Y,  expand=True)
mainloop()
```

Пакування з різними притисканнями у одному вікні

Приклад 9.26

```
from tkinter import *
root=Tk()
button1 = Button(text="1", font = ("Arial", 20))
button2 = Button(text="2", font = ("Arial", 20))
button3 = Button(text="3", font = ("Arial", 20))
button4 = Button(text="4", font = ("Arial", 20))
button1.pack(side='top')
button2.pack(side='bottom')
button3.pack(side='left')
button4.pack(side='right')
root.mainloop()
```

Пакування з притисканнями та розтягуваннями у вікні¶

Приклад 9.27.

```
from tkinter import *
root=Tk()
root.title("Різні притискання")
button1 = Button(text="Верх", font = ("Arial", 20),
bg="red")
button2 = Button(text="Низ", font = ("Arial", 20), bg="red")
button3 = Button(text="Зліва", font = ("Arial", 20),
bg="blue",fg="white")
button4 = Button(text="Справа", font = ("Arial",
20),bg="blue",fg="white")
button1.pack(side='top', fill=Y, expand=True)
button2.pack(side='bottom', fill=Y, expand=True)
button3.pack(side='left',fill=X, expand=True)
```

```
button4.pack(side='right',fill=X, expand=True)
root.mainloop()
```

Огляд опцій пакувальника

anchor – указує точку відліку для урахування розміщення віджета .

За замовчуванням anchor=CENTER

Можливі значення опції anchor:

1) anchor=W – положення «West»

2) anchor=E – положення «East»

3) anchor=N – положення «North»

4) anchor=S – положення «South»

expand (розгортати)

True – віджет заповнює вільний простір

False – (за замовчуванням) віджет не розширюється

Застосування **anchor** показано на прикладі:

Приклад 9.28.

```
from tkinter import *
root=Tk()
button1 = Button(text="1",font = ("Arial", 20))
button2 = Button(text="2",font = ("Arial", 20))
button3 = Button(text="3",font = ("Arial", 20))
button4 = Button(text="4",font = ("Arial", 20))
button1.pack(anchor=N, expand=True)
button2.pack(anchor=W, expand=True)
button3.pack(anchor=E, expand=True)
button4.pack(anchor=S, expand=True)
root.mainloop()
```

fill (заповнювати)

Припустимі значення:

X або - 'x' – заповнювати горизонтально,

Y або 'y' – заповнювати вертикально,

BOTH або '**both**' – в обох напрямках,

NONE або '**none**' – не заповнювати.

padx і **pady**

Відстань, що вказує, який проміжок повинен залишатися ззовні в кожної сторони дочірнього елемента керування.

side (сторона)

Припустимі значення:

LEFT або "left" (ліворуч),

RIGHT або "right" (праворуч),

TOP або "top" (нагорі),

BOTTOM або "bottom" (унизу).

"Внутрішні" елементи керування (дочірні) упаковуються якнайближче до заданої сторони зовнішнього елемента (батька).

Більш докладну інформацію про пакувальника та його опції можна знайти у відповідній документації.

Приклад 9.29. Застосування властивостей padx, pady

```
from tkinter import *
root=Tk()
frame1=Frame(root,bg='purple',bd=5)
lab1 = Label(frame1,text="Північна label",fg="yellow",
bg="green", padx=20,font = ("Arial", 20))
lab2 = Label(root,text="Західна label",fg="white",
bg="blue",pady=20,font = ("Arial", 20))
lab3 = Label(root, text="Східна label",fg="white",
bg="blue",pady=20,font = ("Arial", 20))
lab4 = Label(root,text="Південна label",fg="yellow",
bg="green", padx=20,font = ("Arial", 20))
frame1.pack(anchor=N, expand=True)
```

```
lab1.pack(anchor=N, expand=True)
lab2.pack(anchor=W, expand=True)
lab3.pack(anchor=E, expand=True)
lab4.pack(anchor=S, expand=True)
root.mainloop()
```

Пакувальник **grid()**

Цей пакувальник є таблицею з комірками, у яких містяться віджети.

Синтаксис: **widget.grid(<аргументи>)**

Аргументи

- **row** – номер рядка, у який поміщаємо віджет.
- **rowspan** – скільки рядків займає віджет
- **column** – номер стовпця, у який поміщаємо віджет.

  За замовчуванням `column = 0` (Самий лівий стовпець)
- **columnspan** – скільки стовпців займає віджет.

Приклад 9.30

```
from tkinter import *
root = Tk()
label1 = Label(root, text="example",font = ("Arial", 20))
button1 = Button(root, text = '1',font = ("Arial", 20))
button2 = Button(root, text = '2',font = ("Arial", 20),
ipad=10)
label1.grid(row=0,column=1,columnspan=3,)
button1.grid(row=1,column=0)
button2.grid(row=1,column=4)
root.mainloop()
```

- **padx / pady** – розмір зовнішньої границі (бордюру) по горизонталі й вертикалі.

- **ipadx / ipady** – розмір внутрішньої границі (бордюру) по горизонталі й вертикалі. Різниця між **pad** і **ipad** у тому, що при вказівці **pad** розширюється вільний простір, а при **ipad** розширюється віджет, який поміщаємо.
- **sticky** ("**n**", "**s**", "**e**", "**w**" або їх комбінація) – указує, до якої границі "приклеювати" віджет. Дозволяє розширювати віджет у зазначеному напрямку. Границі названі відповідно до сторін горизонту: "**n**" (північ) – верхня границя, "**s**" (південь) – нижня, "**w**" (захід) – ліва, "**e**" (схід) – права.

Приклад 9.31. Застосування **padx, pady** у пакувальнику **grid**

```
from tkinter import *
root = Tk()
label1 = Label(root, text="example",bg="red",font =
("Arial", 20),relief=SUNKEN)
button1 = Button(root, text = '1',font = ("Arial", 20))
button2 = Button(root, text = '2',font = ("Arial", 20))
label1.grid(row=0,column=1,columnspan=3,padx=50,pady=50)
button1.grid(row=1,column=0)
button2.grid(row=1,column=4)
root.mainloop()
```

Приклад 9.32. Застосування **ipadx, ipady** у пакувальнику **grid**

```
from tkinter import *
root = Tk()
label1 = Label(root, text="example",bg="red",font =
("Arial", 20),relief=SUNKEN)
button1 = Button(root, text = '1',font = ("Arial", 20))
button2 = Button(root, text = '2',font = ("Arial", 20))
label1.grid(row=0,column=1,columnspan=3,ipadx=50,ipady=50)
button1.grid(row=1,column=0)
button2.grid(row=1,column=4)
root.mainloop()
```

Пакувальник **grid()**

Цей пакувальник є таблицею з комірками, у яких містяться віджети.

Синтаксис: **widget.grid(<аргументи>)**

Аргументи

- **row** – номер рядка, у який поміщаємо віджет.
  (row=0 перший рядок)
- **rowspan** – скільки рядків займає віджет
- **column** – номер стовпця, у який поміщаємо віджет.
  За замовчуванням `column = 0` (Самий лівий стовпець)
- **columnspan** – скільки стовпців займає віджет.
- **padx/pady** – розмір зовнішньої границі (бордюру) по горизонталі й вертикалі.
- **ipadx / ipady** – розмір внутрішньої границі (бордюру) по горизонталі й вертикалі. Різниця між **pad** і **ipad** у тому, що при вказівці **pad** розширюється вільний простір, а при **ipad** розширюється віджет, який поміщаємо.
- **sticky** (**"n"**, **"s"**, **"e"**, **"w"** або їх комбінація) – указує, до якої границі "приклеювати" віджет. Дозволяє розширювати віджет у зазначеному напрямку. Границі названі відповідно до сторін горизонту: **"n"** (північ) – верхня границя, **"s"** (південь) – нижня, **"w"** (захід) – ліва, **"e"** (схід) – права.

Приклад 9.33. Вплив зміни параметра **sticky**

```
from tkinter import *
root = Tk()
label1 = Label(root, text="example",font = ("Arial", 20))
button1 = Button(root, text = '1',font = ("Arial", 20))
button2 = Button(root, text = '2',font = ("Arial", 20))
button3 = Button(root, text = '3',font = ("Arial", 20))
button4 = Button(root, text = '4',font = ("Arial", 20))
label1.grid(row=0,column=0,columnspan=3)
button1.grid(row=1,column=2)
```

```
button2.grid(row=1,column=4)
button3.grid(row=2,column=2, sticky = "e")
button4.grid(row=2,column=4)
root.mainloop()
```

Додаткові методи

`grid_configure` – синонім для **`grid`**.

`grid_slaves -` повертає список віджетів, що належать даному віджетові. Віджети повертаються у вигляді посилань віджета `tkinter`.

`grid_size` – повертає розмір таблиці в рядках і стовпцях.

`grid_info -` повертає словник, що містить поточні параметри для середовища, яке оточує цей віджет.

`grid_location` – приймає два аргументи: ***`x`*** і ***`y`*** (у пікселях). Повертає номер рядка й стовпця, в які потрапляють зазначені координати, або *-1,* якщо координати потрапили поза віджет.

Приклад 9.34. Використання додаткових методів

```
from tkinter import *
root = Tk()
label1 = Label(root, text="example",font = ("Arial", 20))
button1 = Button(root, text = '1',font = ("Arial", 20))
button2 = Button(root, text = '2',font = ("Arial", 20))
label1.grid(row=0,column=1,columnspan=3)
button1.grid(row=1,column=0, sticky = "w")
button2.grid(row=1,column=4)
print("Розмір сітки: ",root.grid_size())
print("Сітка містить: ",root.grid_slaves())
print("Параметри комірки: ", label1.grid_info())
print("Комірка: ", root.grid_location(150,100))
root.mainloop()
```

Методи налаштування комірок

Налаштування параметрів для стовпця комірки.

Формат:

<root>.**columnconfigure (індекс,<опції>)**

індекс – індекс стовпця решітки.

<опції>

minsize=<число>

Визначає мінімальний розмір стовпця. Якщо стовпець повністю порожній, він не буде відображатися, навіть якщо ця опція встановлена.

weight= =<число>

Відносна вага використовується для розподілу додаткового простору між стовпцями. Стовпець з вагою 2 зростатиме вдвічі швидше, ніж стовпець з вагою 1. Значення за замовчуванням дорівнює 0, що означає, що стовпець не буде рости взагалі.

Налаштування параметрів для рядка комірки.

Формат:

<root>.**rowconfigure (індекс,<опції>)**

індекс – індекс рядка решітки.

<опції>

minsize=<число>

Визначає мінімальний розмір рядка. Якщо рядок повністю порожній, він не буде відображатися, навіть якщо ця опція встановлена.

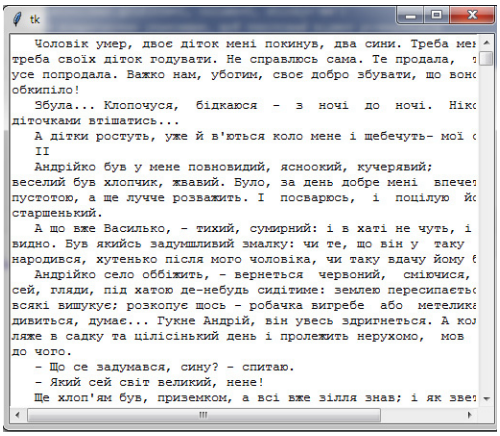
weight= =<число>

Відносна вага використовується для розподілу додаткового простору між рядками. Рядок з вагою 2 зростатиме вдвічі швидше, ніж рядок з вагою 1. Значення за замовчуванням дорівнює 0, що означає, що рядок не буде рости взагалі.

Приклад 9.35. Текстовий віджет з двома скролбарами

```
from tkinter import *
f=open("1.txt")
data = f.read()
```

```
f.close
root=Tk()
text = Text(wrap=NONE)
text.insert(END,data)
vscrollbar = Scrollbar(orient='vert', command=text.yview)
text['yscrollcommand'] = vscrollbar.set
hscrollbar = Scrollbar(orient='hor', command=text.xview)
text['xscrollcommand'] = hscrollbar.set
# розміщуємо віджети
text.grid(row=0, column=0, sticky='nsew')
vscrollbar.grid(row=0, column=1, sticky='ns')
hscrollbar.grid(row=1, column=0, sticky='ew')
# конфігуруємо упаковщик, щоб текстовий віджет розширювався
root.rowconfigure(0, weight=1)
root.columnconfigure(0, weight=1)
root.mainloop()
```



vscrollbar

hscrollbar

Приклад 9.36. Розміщає 12 лейблів у сітці розміром 3x4.

```
import tkinter
root = tkinter.Tk()
for r in range(3):
    for c in range(4):
```

```
            lb = tkinter.Label(root, text='R%s/C%s'%(r,c), bd=10, font = ("Arial", 20))
            lb.grid( row=r, column=c)
            root.rowconfigure(c, weight=1)
        root.columnconfigure(r, weight=1)
print("Сітка містить: ",root.grid_slaves())
root.mainloop(  )
```

Приклад 9.37. Вікно введення даних про студента

```
from tkinter import *
root = Tk( )
l1=Label(root, text="Група", font = ("Arial", 20))
l1.grid(row=0, sticky=W) # за замовчуванням column=0
l2=Label(root, text="П.І.Б.", font = ("Arial", 20))
l2.grid(row=1,sticky=W) # за замовчуванням column=0
l3=Label(root, text="№", font = ("Arial", 20))
l3.grid(row=2, sticky=W) # за замовчуванням column=0
e1 = Entry(root, font = ("Arial", 20))
e2 = Entry(root, font = ("Arial", 20))
e3 = Entry(root,font = ("Arial", 20))
e1.grid(row=0, column=1, sticky="ew")
e2.grid(row=1, column=1, sticky="ew")
e3.grid(row=2, column=1, sticky="ew")
root.mainloop()
```

Використання опцій злиття стовпців або рядків

Опції **`columnspan`** *та* **`rowspan`**

Ці опції дозволяють виділяти під віджети більше, ніж одну комірку.

Опція **`columnspan=<число>`** опція використовується для виділення під один віджет більше одного стовпця.

Опція **rowspan=<число>** опція використовується для виділення під один віджет більше одного рядка.

Приклад 9.38.

```
from tkinter import *
root = Tk( )
l1=Label(root, text="Group",font='arial 20')
l1.grid(sticky=E)
l2=Label(root, text="Name",font='arial 20')
l2.grid(sticky=E)
l3=Label(root, text="№",font='arial 20')
l3.grid(sticky=E)
button1=Button(root,text='ok',width=3,bg='red',fg='black', font='arial 20')
button2=Button(root,text='no',width=3,bg='black',fg='red', font='arial 20')
p = PhotoImage(file="Tux.gif")
iml = Label(root, image=p)
e1 = Entry(root,font='arial 20')
e2 = Entry(root,font='arial 20')
e3 = Entry(root,font='arial 20')
e1.grid(row=0, column=1, sticky="ew")
e2.grid(row=1, column=1, sticky="ew")
e3.grid(row=2, column=1, sticky="ew")
cb = Checkbutton(root,text="Life's good",font='arial 20')
cb.grid(columnspan=2, sticky=W)
iml.grid(row=0, column=2, columnspan=2, rowspan=2, sticky=W+E+N+S, padx=5, pady=5)
button1.grid(row=2, column=2)
button2.grid(row=2, column=3)
root.mainloop( )
```

Пакувальник **place**

Пакувальник **place** є найпростішим із трьох загальних менеджерів геометрії, передбачених в **tkinter**.

Він дозволяє точно встановити положення й розмір вікна або в абсолютному значенні, або відносно іншого вікна.

Це, як правило, не дуже гарна ідея – використовувати `Place` для звичайних і діалогових вікон.

Просто прийдеться виконати багато роботи, щоб усе запрацювало як годиться. Рекомендується використовувати `pack` або `grid` для таких цілей.

Синтаксис:

widget.place(<place_options>)

Основні опції пакувальника `place`

- **anchor:** Точка відліку для розміщення віджета встановлюється буквами по сторонах горизонту:
- **N** – північ,
- **E** – схід,
- **S** – південь,
- **W** – захід,
- **NE** – північний схід,
- **NW –** північний захід,
- **SW** – південний захід, або
- **NW -** значення за замовчуванням: (верхній лівий кут батьківського віджета).
- **bordermode:**

INSIDE (за замовчуванням) для індикації того, що враховують розташування усередині (ігноруючи розташування ззовні);

OUTSIDE – навпаки.

- **height, width:** висота й ширина в пікселях.
- **relheight, relwidth:** висота й ширина виражена дробом від 0.0 до 1.0, як частка від висоти й ширини батьківського віджета.

- **relx, rely:** горизонтальний і вертикальний зсув як дріб між 0.0 і 1.0, у частках від висоти й ширини батьківського віджета.
- **x, y:** горизонтальний і вертикальний зсув у пікселах.

Приклади роботи пакувальника **place**

Наступна команда центрує віджет усередині батька.

```
w.place(relx=0.5, rely=0.5, anchor=CENTER)
```

Скористаємося цією властивістю для розміщення кнопки в центрі батьківського об'єкта `root`.

Приклад 9.39.

```
from tkinter import *
root = Tk()
def helloCallBack():
   print( "Hello Python")
B = Button(root, height=2, width=10, text ="Hello", command
= helloCallBack)
B.place( relx=0.5, rely=0.5,anchor=CENTER,
relheight = 0.35, relwidth=0.35)
root.mainloop()
```

Пояснення роботи прикладу

Віджет міститься на середині довжини й висоти батьківського об'єкта **root**. Точка відліку для віджета **Button** узята на його середині, оскільки `anchor=CENTER`. При натисканні на кнопку опція `command` викликає функцію `hellocallback()`,яка видає у вихідний потік `stdout` повідомлення **"Hello Python"**

Властивості `relheight = 0.2, relwidth=0.35` задають висоту та ширину віджета, як частку від відповідних параметрів віджета, який є власником даного віджета.

В **root** додали ще один об'єкт **Label.** Тепер потрібно самостійно стежити, щоб зображення цих об'єктів не перетиналися.

Приклад 9.40.

```
from tkinter import *
root = Tk()
root.geometry("300x300")
def helloCallBack():
   print("Hello Python")
B = Button(root, height=2, width=10, text ="Hello", command
= helloCallBack)
L = Label(root, text= "Welcome button", fg="red",)
B.place( relx=0.5, rely=0.5,anchor=CENTER)
L.place( relx=0.5, rely=0.25,  anchor=S)
root.mainloop()
```

Застосування пакувальника **place** *при використанні віджета* **Frame**

Розглянемо використання віджета `Frame`, який є контейнером для інших віджетів. У цьому випадку батьком віджета `Frame` є `root`.

А батьком віджетів `Label` і `Button` є віджет `Frame`.

Усі об'єкти впаковані за допомогою пакувальника (менеджера геометрії) `place`.

Хоча для цих цілей можна використовувати як `pack`, так і `grid`

Приклад 9.41.

```
from tkinter import *
root = Tk()
```

```
pane=Frame(root, height=150, width=150, bg = "green", relief
= RAISED, bd = 10)
pane.place(relx=0.5,rely=0.5,anchor=CENTER)
def helloCallBack():
   print( "Hello Python")
B = Button(pane, height=2, width=10, text ="Hello", command
= helloCallBack)
L = Label(pane, text= "Welcome button", fg="red",bg =
"yellow")
B.place( relx=0.5, rely=0.5,anchor=CENTER)
L.place( relx=0.5, rely=0.25,  anchor=S)
root.mainloop()
```

message box

Модуль **messagebox** використовується для відображення вікон повідомлень у додатках.

Модуль **messagebox** забезпечує ряд функцій, які можна використовувати, щоб відобразити відповідне повідомлення.

Синтаксис:

```
<змінна> = messagebox.functionname(<"заголовок">,
<"повідомлення">, [, опції])
```

functionname – ім'я відповідної функції `messagebox`.

`"заголовок"` - Текст, який відображається в смузі заголовка вікна повідомлення.

"повідомлення" - текст, який відображається як повідомлення.

опції - необов'язковий параметр для налаштування конфігурації вікна й виду кнопки.

Функції для діалогового вікна (functionname):

- `showinfo()`
- `showwarning()`
- `showerror ()`

- `askquestion()`
- `askokcancel()`
- `askyesno ()`
- `askretrycancel()`

Для виклику цього модуля не достатньо інструкції

```
from tkinter import *
```

Необхідно додатково записати окремий виклик модуля

```
from tkinter import messagebox
```

Виклик діалогового вікна з пакувальником **`place()`**

Приклад 9.42.

```
from tkinter import *
from tkinter import messagebox
root = Tk()
def hello():
   messagebox.showinfo("Say Hello", "Hello World")
B1 = Button(root, text = "Say Hello", command = hello)
B1.place(relx=0.5, rely=0.25,  anchor=S)
mainloop()
```

Використання двох кнопок, розміщених на **`Frame`***, і пакувальника* **`grid`**

Приклад 9.43.

```
from tkinter import *
from tkinter import messagebox
root = Tk()
app = Frame(root, bg = "green" , bd=10)
app.grid(row=0, column = 0)
def dialog():
   var = messagebox.showinfo("test" , "This is my first message")
button2 = Button(app, text = "Info" , width=5, command=dialog)
button2.grid(padx=110, pady=80)
```

```
button1 = Button(app, text = " exit " , width=3, command=exit)
button1.grid(row=1, column = 2)
mainloop()
```

9.1.6. Застосування меню

Основні функціональні можливості віджета:

- спливаюче вікно (**pop-up**),
- меню верхнього рівня (**toplevel**),
- меню, що випадає (**pull-down**).

Синтаксис.

```
<змінна> = Menu(<батьківське вікно>, <опція>, ... )
```

<батьківське вікно> – представляє батьківське вікно. Найбільше часто – це `root`.

<опція> – приведемо список найбільш часто використовуваних опцій для цього віджета. Ці параметри можуть бути використані як пари ключ-значення, розділені комами.

Опції	Опис
`activebackground`	Колір тла, яким буде відображатися вибір, коли він перебуває під мишкою.
`activeborderwidth`	Визначає ширину границі, проведеної навколо вибору, коли він перебуває під мишкою. За замовчуванням 1 піксель.
`activeforeground`	Колір переднього плану, яким буде відображатися вибір, коли він перебуває під мишкою.
`bg`	Колір тла для вибору не під мишкою.
`bd`	Ширина границі навколо всіх варіантів. Значення за замовчуванням 1.

Опції	Опис
cursor	Курсор, який з'являється при наведенні курсору миші на вибір, але тільки тоді, коли меню виключене.
disabledforeground	Колір тексту для елементів у стані DISABLED.
font	Шрифт за замовчуванням для текстових варіантів.
fg	Колір переднього плану використовується для варіантів вибору, що не перебувають під мишею.
postcommand	Ви можете встановити цю опцію в процедуру, і ця процедура буде викликатися щоразу, коли хтось відвідує це меню.
relief	3 D-Ефект за замовчуванням для меню при relief=RAISED.
image	Для виводу зображення на цій кнопці меню.
selectcolor	Визначає колір, відображуваний в checkbutton, коли вони обрані.
tearoff	Меню може бути відокремлене, перша позиція (позиція 0) у списку вибору займає головний елемент (tearoff), а додаткові варіанти будуть додані, починаючи з позиції 1. Якщо ви встановили tearoff=0, меню не буде мати функцію tearoff, і вибір буде доданий, починаючи з позиції 0.
Title	Як правило, заголовок вікна меню tearoff буде таким же, як текст кнопки меню каскаду, який приводить до цього меню. Якщо ви прагнете змінити назву цього вікна, установіть параметр заголовка на цей рядок.

Методи, що доступні на об'єктах меню:

Методи	Опис
`add_command (options)`	Додає пункт меню в меню.
`add_radiobutton(options)`	Створює елемент меню `radio button`.
`add_checkbutton(options)`	Створює пункт меню `checkbutton`.
`add_cascade(options)`	Створює нове ієрархічне меню, зв'язуючи дане меню з батьківським меню.
`add_separator()`	Додає роздільник у меню.
`add(type, options)`	Додає певний тип пункту меню в меню.
`delete(startindex [, endindex])`	Видаляє пункти меню, починаючи від `startindex` до `endindex`.
`entryconfig(index, options)`	Дозволяє змінити пункт меню, який ідентифікується за допомогою індексу, і змінити його параметри.
`index(item)`	Повертає порядковий номер даного лейбла пункту меню.
`insert_separator (index)`	Включити новий роздільник у позиції, зазначеній індексом.
`invoke (index)`	Викликає команду зворотного виклику, пов'язаного з вибором позиції індексу.
`type (index)`	Повертає тип вибору зазначеного індексу: `"cascade"`, `"checkbutton"`, `"command"`, `"radiobutton"`, `"separator"` або `"tearoff"`.

Розглянемо роботу меню на прикладі

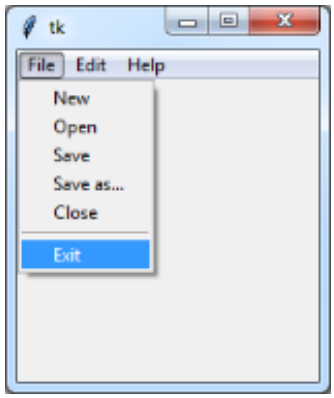
Приклад 9.44.

```
from tkinter import *

def donothing():
```

```
    filewin = Toplevel(root)
    button = Button(filewin, text="Do nothing button")
    button.pack()
root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)
editmenu.add_separator()
editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
editmenu.add_command(label="Select All", command=donothing)
menubar.add_cascade(label="Edit", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)
```

root.config(menu=menubar)

root.mainloop()



Menubutton

Синтаксис

<змінна> = Menubutton (<батьківське вікно>, <опція>, ...)

<батьківське вікно> – Представляє батьківське вікно. Найбільше часто – це `root`.

<опція> – приведемо список найбільше часто використовуваних опцій для цього віджета.

Ці параметри можуть бути використані в якості пар ключ-значення, розділених комами.

Опції	Опис
`activebackground`	Колір тла при наведенні курсору миші на `menubutton`.
`activeforeground`	Колір переднього плану, коли миша перебуває над `menubutton`.
`anchor`	Цей параметр визначає, де текст позиціонується, якщо віджет має більше простору, ніж потрібно для тексту. За замовчуванням `anchor=CENTER`, який центрує текст.
`bg`	Нормальний колір тла, відображуваного за лейблом і індикатором.

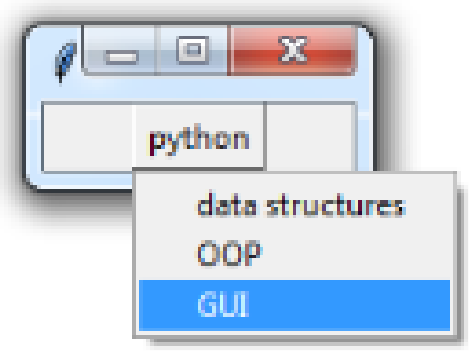
Опції	Опис
bitmap	Для відображення растрового зображення на кнопку MENU установіть цей параметр **на ім'я растрового зображення**.
bd	Розмір границі навколо індикатору. За замовчуванням 2 пікселя.
cursor	Курсор, який з'являється при наведенні курсору миші на menubutton.
direction	Установите direction=LEFT, щоб відобразити меню ліворуч від кнопки; використовуйте direction=RIGHT, щоб відобразити меню праворуч від кнопки; або напрямок direction='above', щоб помістити меню над кнопкою.
disabledforeground	Колір переднього плану в ситуації, коли menubutton відключений.
fg	Колір переднього плану, коли миша не перебуває над menubutton.
height	Висота menubutton у рядках тексту (не пикселах). За замовчуванням відповідає розміру тексту в menubutton.
highlightcolor	Колір у фокусі підсвічування, коли віджет має фокус.
image	Для виводу зображення на menubutton
justify	Цей параметр визначає, де текст розташований, коли текст не заповнює menubutton: використовувати ustify=LEFT для розміщення зліва (за замовчуванням),

Опції	Опис
	використовувати `ustify=CENTER` для центрування, або `ustify=RIGHT` направо.
`menu`	Щоб зв'язати `menubutton` з набором варіантів, установіть цей параметр в об'єкт `Menu`, що містить ці набори. Цей об'єкт `Menu` повинен бути створений шляхом передачі зв'язаної `menubutton` конструктору як перший аргумент.
`padx`	Скільки місця залишити ліворуч і праворуч від тексту `menubutton`. Значення за замовчуванням 1.
`pady`	Скільки місця залишити вище й нижче тексту `menubutton`. Значення за замовчуванням 1.
`relief`	Вибирає тривимірні пограничні ефекти затінення. За замовчуванням піднята.
`state`	Зазвичай `menubuttons` реагує на мишу. Установіть `state=DISABLED`, щоб зробити `menubutton` сірою. У цьому стані вона не буде відповідати на запити.
`text`	Для відображення тексту на menubutton установіть в цей параметр рядок, що містить потрібний текст. Знак нового рядка ("\n") усередині рядка буде викликати розриви рядків.
`textvariable`	Можна зв'язати змінну управління класу `Stringvar` із цим `menubutton`. Встановлення цієї змінної керування змінить відображуваний текст.

Опції	Опис
`underline`	Зазвичай в `menubutton` текст відображається без підкреслення. Щоб підкреслити один із символів, установіть для цього параметра індекс цього символу.
`width`	Ширина віджета в символах. Значення за замовчуванням 20.
`wraplength`	Як правило, рядки не переносяться. Ви можете встановити цю опцію на ряд (низку) символів, і всі рядки будуть розбиті на куски не довші, ніж це число.

Приклад 9.45.

```
from tkinter import *
top = Tk()
mb=  Menubutton ( top, text="python", relief=RAISED )
mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0 )
mb["menu"]  =  mb.menu
datVar = IntVar()
oopVar = IntVar()
guiVar = IntVar()
mb.menu.add_checkbutton ( label="data structures", variable=datVar )
mb.menu.add_checkbutton ( label="OOP",variable=oopVar )
mb.menu.add_checkbutton ( label="GUI", variable=guiVar )
mb.pack()
top.mainloop()
```



Toplevel

Віджети верхнього рівня працюють як вікна, які безпосередньо керуються диспетчером вікон. Вони не обов'язково мають батьківський віджет над ними зверху.

Наша програма може використовувати будь-яку кількість вікон верхнього рівня.

Синтаксис.

```
<Змінна> = Toplevel( <опція>, ... )
```

<опція>: нижче наведений список найчастіше використовуваних опцій для цього віджета. Ці параметри можуть бути використані як пари ключ-значення, розділені комами.

Опція	Опис
`bg`	Колір тла вікна.
`bd`	Ширина рамки в пікселях; за замовчуванням дорівнює 0.
`cursor`	Курсор, який з'являється при наведенні курсору миші в цьому вікні.
`font`	Шрифт за замовчуванням для тексту усередині віджета.
`fg`	Колір, використовуваний для тексту (і растрові зображення) у межах віджета. Ви можете змінити колір для мічених регіонів; ця опція тільки за замовчуванням.
`height`	Висота вікна.
`relief`	Як правило, вікно верхнього рівня не буде мати 3-d границі довкола нього. Щоб одержати тінисту границю, установіть опцію

	bd більше, ніж її нульове значення за замовчуванням, і встановіть опцію `relief` у константу.
`width`	Необхідна ширина вікна.

`Toplevel` *об'єкти мають такі методи*

Методи і їх опис
`frame()` Повертає системний ідентифікатор вікна.
`state()` Повертає поточний стан вікна. Можливі значення `normal`, `zoommed`, `withdrawn` and `icon`
`withdraw()` Видаляє вікно з екрана, не руйнуючи його.
`maxsize(width, height)` Визначає максимальний розмір вікна.
`minsize(width, height)` Визначає мінімальний розмір вікна.
`resizable(width, height)` Визначає зміну розміру прапорів, які управляють, чи можна змінювати розміри вікна.
`title(string)` Визначає заголовок вікна.

Приклад 9.46.

```
from tkinter import *
root = Tk()
root.title("Root window")
top = Toplevel(root, height=500, width=500)
top.title("My first toplevel window")
top.mainloop()
```

Canvas

Canvas (полотно) являє собою прямокутну площу, призначена для малювання. На canvas можна розмістити графіку, текст, віджети та фрейми.

Синтаксис.

<**змінна**>=**Canvas(<батьківський об'єкт>,**

<опція>=value,...)

<батьківське вікно> – представляє батьківське вікно. Найчастіше – це **root.**

<опція>: Нижче наведений список найчастіше використовуваних опцій для цього віджета. Ці параметри можуть бути використані в якості пар ключ-значення, розділених комами.

Опції	Опис
bd	Ширина рамки в пікселях. За замовчуванням дорівнює 2.
bg	Нормальний колір тла.
confine	Якщо True (за замовчуванням), canvas не може прокручуватися за межами scrollregion.
cursor	Курсор, що використовується в полотні, як arrow, circle, dot і т.д.
height	Розмір canvas у напрямку Y.
highlightcolor	Колір у фокусі підсвічування.
relief	Рельєф визначає тип границі. Деякі зі значень SUNKEN, RAISED, GROOVE і RIDGE.
scrollregion	Кортеж (w, n, e, s), який визначає те, яку область canvas можна прокручувати, де w (захід) - це ліва сторона, n (північ)-верх, e (схід) - права сторона, і s (південь) – нижня сторона.
Width	Розмір canvas у напрямку X.

Canvas віджет може підтримувати наступні стандартні елементи:

arc (дуга). Створює елемент дуги, який може бути хордою, круговою діаграмою або простою дугою.

Приклад 9.47.

```
coord  = (10, 50, 240, 210)
```

coord – це координати обмежуючого блока для всієї дуги.

```
arc = canvas.create_arc(coord, start=90, extent=150,
fill="blue")
```

Створення об'єкта дуги в заданих координатах **coord.**

start=90 – початковий кут. За замовчуванням 0.0

extent=150 – кінцевий кут відносно початкового кута.

fill="green" – колір заповнення

image (зображення). Створює елемент зображення, який може бути екземпляром класів **BitmapImage** або **PhotoImage.**

```
filename = PhotoImage(file = "sunshine.gif")
image = canvas.create_image(400, 0, anchor=NE,
image=filename)
```

400,0 – точка установки зображення

anchor=NE – точка відліку (північно-східний кут)

image=filename – об'єкт типу **PhotoImage**, що містить попередньо завантажене зображення з файла "**sunshine.gif**"

line (лінія). Створює елемент лінію.

```
line = canvas.create_line(x0, y0, x1, y1, ..., xn, yn,
options)
```

x0, y0 координати початкової точки,

x1, y1 координати наступної точки,

xn, yn координати наступної точки.

options – опції, які задають вигляд лінії, наприклад:

fill=**"blue"** лінія голуба

width =2 товщина лінії 2 пікселя

oval (овальна форма). Створює коло або еліпс у заданих координатах. Потрібно дві пари координат; верхній лівий і нижній правий кути прямокутника для овалу.

```
oval = canvas.create_oval(x0, y0, x1, y1, options)
```

`x0, y0` – координати верхнього лівого кута обмежуючого прямокутника.

`x1, y1` – координати нижнього правого кута обмежуючого прямокутника.

`options` –опції зовнішнього вигляду, наприклад:

`fill="blue"` колір заповнення,

`width =10` товщина обмежуючої лінії

polygon (багатокутник). Створює полігон, який повинен мати принаймні три вершини.

```
oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn,
options)
```

`x0, y0` координати початкового кута**.**

`x1, y1` координати наступного кута**.**

`xn, yn` координати кінцевого кута**.**

`options` –опції зовнішнього вигляду, наприклад:

`fill="red"` колір заповнення.

Приклад 9.48.

```
from tkinter import *
top = Tk()
C = Canvas(top, bg="white", height=250, width=500)
coord = (10, 50, 240, 210)
arc = C.create_arc(coord, start=90, extent=150, fill="green")
filename = PhotoImage(file = "sunshine.gif")
image = C.create_image(500, 0, anchor=NE, image=filename)
line = C.create_line(20, 20, 100, 20, fill="blue", width =2)
ov= C.create_oval(30, 30, 100, 100, fill="blue", width =10)
plg = C.create_polygon(110, 180, 200, 220, 180, 30, fill="red")
```

```
C.pack()
top.mainloop()
```

Віджет **PanedWindow**

`PanedWindow` **-** контейнерний віджет, який містить певну кількість панелей, розташованих по горизонталі або по вертикалі.

Кожна містить віджет і кожна пара панелей відділяється пересувним роздільником. Переміщення роздільника викликає зміну розмірів віджетів.

Синтаксис.

<**змінна**> = **PanedWindow (<батьківське вікно>, <опція>,...)**

<батьківське вікно> – представляє батьківське вікно. Найчастіше – це `root`.

<опція>: нижче наведений список найбільш часто використовуваних опцій для цього віджета. Ці параметри можуть бути використані як пари ключ-значення, розділені комами.

Опції	Опис
`bg`	Колір слайдера й наконечника, коли миша не перебуває над ними.
`bd`	Ширина 3D границі по всьому периметру,.
`borderwidth`	За замовчуванням дорівнює 2.
`cursor`	Курсор, який з'являється при наведенні курсору миші над вікном.
`height`	Значення за замовчуванням відсутнє.
`orient`	За замовчуванням горизонтальна.
`relief`	За замовчуванням є плоскою.
`sashcursor`	Значення за замовчуванням відсутнє.
`sashrelief`	За замовчуванням піднята.
`sashwidth`	За замовчуванням дорівнює 2.
`showhandle`	Немає значення за замовчуванням.
`width`	Значення за замовчуванням відсутнє.

Приклад 9.49.

```
from tkinter import *
m1 = PanedWindow(bd=1, bg="black",height=300,width=300)
m1.pack(fill=BOTH, expand=1)
left = Label(m1, text="left pane", font = "Arial 18")
m1.add(left)
m2 = PanedWindow(m1, orient=VERTICAL, bd=1,bg="red",)
m1.add(m2)
top = Label(m2, text="top pane", font = "Arial 18")
m2.add(top)
bottom = Label(m2, text="bottom pane",font = "Arial 18")
m2.add(bottom)
mainloop()
```

9.1.7. Події, зв'язування і опитування

Як уже згадувалося раніше, програма з використанням **tkinter** проводить більшу частину свого часу усередині циклу обробки подій (за допомогою методу **`mainloop`**).

Джерела подій:

-натискання клавіш на клавіатурі,

-дії з мишею (ліва кнопка, права кнопка, протягування)

-перенаправлення від менеджера вікон.

Приклад обробки подій:

`callbacks` (зворотні виклики), виконувані, за допомогою `collables` – об'єктів, встановлюваних за допомогою команди `command=<callable oject>`

`tkinter` не дозволяє створювати свої власні події; ви обмежені роботою з подіями, визначеними самим `tkinter`.

Об'єкт **`Event`**

У загальному випадку функції зворотного виклику для події повинні приймати одну подію-аргумент, яка є об'єктом події `tkinter`. Такий об'єкт події має кілька атрибутів, що описують подію:

`widget-` віджет, що генерує цю подію. Це діючий екземпляр віджета з модуля `tkinter`, а не ім'я. Цей атрибут установлюється для всіх подій.

`x, y-` поточне положення миші, у пікселях.

`x_root, y_root -` поточне положення миші відносно верхнього лівого кута екрана в пікселях.

`char-`рядок з одного символу, що є кодом натиснутої кнопки (тільки для подій від клавіатури).

`keysym –` рядок, що є символічним іменем кнопки клавіатури (тільки для подій клавіатури).

`keycode -` код кнопки клавіатури (тільки події клавіатури).

`num -` номер кнопки (тільки кнопка події миші).

`width, height -` новий розмір віджета, у пікселях

`type -` тип події.

Метод зв'язування **`bind()`**

`tkinter` забезпечує потужний механізм самостійної обробки подій.

Для кожного віджета можна зв'язати функції й методи мови `Python` з подіями.

Синтаксис.

***`w`*`.bind(`*`event_name`*`,`*`callback`*`)`**

Якщо подія, відповідна до опису, відбувається у віджеті, даний ***`callback`*** (так називають функцію зворотного виклику) викликається з параметром-об'єктом, який описує подію.

Приклад 9.50. Захоплення «кліку» на вікні

```
from tkinter import *
root = Tk()
def callback(event):
```

```
    print("clicked at", event.x, event.y)
frame = Frame(root, width=200, height=200)
frame.bind("<Button-1>", callback)
frame.pack()
root.mainloop()
```

У цьому прикладі ми використовуємо метод **bind** для віджета **Frame**, щоб зв'язати функцію **callback** з подією за назвою **<Button-1>.** Якщо запустити програму й натискати на вікні, що з'явилося, ліву кнопку миші, то при кожному натисканні в stdout, буде виводитися повідомлення типу **"clicked at 53 30** ".

Події клавіатури

Події клавіатури приходять на віджет, якому в поточний момент належить фокус клавіатури. Можемо використовувати метод **focus_set** для переміщення фокуса на потрібний віджет:

Приклад 9.51. Захоплення подій від клавіатури.

```
from tkinter import *
root = Tk()
def my_key(event):
    print("pressed", repr(event.char))
def callback(event):
    frame.focus_set()
    print("clicked at", event.x, event.y)
frame = Frame(root, width=100, height=100)
frame.bind("<Key>", my_key)
frame.bind("<Button-1>", callback)
frame.pack()
root.mainloop()
```

Пояснення до прикладу

1. Потрібно спочатку «клікнути» по об'єкту **Frame**, що забезпечить встановлення фокуса. Тоді він почне одержувати які-небудь події клавіатури.

2. Натискання лівою кнопкою мишки спричинить подію "<Button-1>", яка викличе функцію callback(event), яка встановить фокус на віджет **frame** та виведе у **stdout** повідомлення з координатами «кліка».

3. При натисканні клавіші на клавіатурі виникає подія **"<Key>"**, яка викличе функцію `key(event)`, що виводить значення атрибуту `event.char`. Цей атрибут містить символ нажатої клавіші.

Події

Події представлені у вигляді рядків, що використовують спеціальний синтаксис подій:

<modifier-type-detail>

Поля **<modifier>** і **<detail>** використовуються, щоб надати додаткову інформацію

(в багатьох випадках можуть не використовуватися)

Поле **type** визначає тип події, яку ми бажаємо зв'язати. Це можуть бути:

- дії користувача, такі як **Button,** і **Key**,

- події менеджера вікон: **Enter**, **Configure** і інші.

Розглянемо найпоширеніші формати подій.

Формати подій

<Button-1>, <Button-2>, <Button-3>

Кнопка миші натиснута над віджетом:

Button-1 – це ліва кнопка миші,

Button-2 – середня кнопка (якщо така є),

Button-3 – права кнопка миші.

При натисканні на кнопку миші на віджеті `tkinter` буде автоматично "захоплювати" курсор миші, і наступні події миші (наприклад, руху) будуть

відправлятися на поточний віджет доти, поки кнопка миші втримується в натиснутому стані, навіть якщо миша переміщається за межі поточного віджета.

Поточне положення курсора миші (відносно віджета) визначають в атрибутах **x** та **y** параметра об'єкта події переданого у функцію зворотного виклику.

<B1-motion>, <B2-motion>, <B3-motion>

Миша переміщається, при натиснутій кнопці миші:

<B1-motion> - натиснута `Button 1`

<B2-motion> - натиснута `Button 2`

<B3-motion> - натиснута `Button 3`

Поточне положення курсора миші знаходиться в **x** та **y** атрибутах об'єкта події, переданого функції зворотного виклику.

<Buttonrelease-1>,<Buttonrelease-2>,<Buttonrelease-3>

<Buttonrelease-1>-`Button 1` була відпущена.

<Buttonrelease-2>-`Button 2` була відпущена.

<Buttonrelease-3>-`Button 3` була відпущена.

Поточне положення курсора миші знаходиться в x та y атрибутах об'єкта події, переданого функції зворотного виклику.

<Double-button-1>,<Double-button-2>,<Double-button-2>

<Double-button-1> - подвійний «клік» на `Button-1`.

<Double-button-2> - подвійний «клік» на `Button-1`.

<Double-button-3> - подвійний «клік» на `Button-1`.

Можна використовувати подвійний **Double** або потрійний **Triple** як префікси.

Якщо зв'язати одночасно один «клік» (`<Button-1>`) і подвійний «клік», то обидві прив'язки будуть спрацьовувати.

<Enter>

Курсор миші ввійшов у віджет (ця подія не означає, що користувач натиснув клавішу `Enter`!).

<Leave>

Курсор миші покинув віджет.

<Focusin>

Фокус клавіатури був перенесений на цей віджет, або на віджети, які йому належать.

<Focusout>

Фокус клавіатури був перенесений із цього віджета на інший віджет.

<Return>

Користувач натиснув клавішу `Enter`. Ви можете зв'язати віртуально всі клавіші на клавіатурі. Для звичайної 102-клавишної клавіатури Pc-стилю,

Спеціальні клавіші

`Alt_L` `Клавіша Alt`	`Cancel` `Клавіша Break`	`Backspace`
`Control_L` `Клавіша Ctrl`	`Prior (Page Up)`	`Pause`
`Shift_L` `Клавіша Shift`	`Next (Page Down)`	`Caps_Lock`
`Num_Lock i` `Scroll_Lock`	`Left, Up, Right, Down`	`End`
`Return` `(Enter)`	`Escape` `Insert`	`Home` `Delete`
`F1,F2,F3,F4,F5,F6,F7,F8,F9,F10,F11,F12`		

<Key> - користувач натиснув будь-яку клавішу. Яка конкретно кнопка була натиснута повертається в об'єкт події у вигляді символу (для спеціальних клавіш-порожній рядок).

<a> - користувач вводить "a".

<Shift-Up> - користувач натиснув *клавішу зі стрілкою нагору*, утримуючи натиснутою клавішу `Shift`. Можна використовувати як префікси `Alt, Shift i Control.`

<Configure> - віджет змінив розмір (або місце розташування, на деяких платформах). Новий розмір передбачений в атрибутах ширини й висоти повернутого об'єкта події.

Екземпляри й зв'язування класів

Метод **bind,** який ми використовували в наведеному вище прикладі, створює екземпляр зв'язування.

Це означає, що прив'язка відбувається тільки до одного віджета; якщо ви створюєте нові фрейми, вони не будуть успадковувати прив'язки.

Але **tkinter** також дозволяє створювати прив'язки до класу та до рівня програми.

Рівні прив'язки подій

Можна створити прив'язки на чотирьох різних рівнях:

1.На рівні екземпляра віджета, використовуючи **bind**.

2.На рівні вікна верхнього рівня віджета (**toplevel** або **root**), також використовуючи **bind**.

3.На рівні класу віджета, використовуючи **bind_class** (це використовується `tkinter` для забезпечення стандартних прив'язок).

4.На рівні програми, використовуючи **bind_all**.

Приклади прив'язки

Можна використовувати **bind_all**, щоб створити прив'язку для клавіші **F1**, так що буде можливість викликати help скрізь у програмі.

Поведінка програми, якщо існує кілька прив'язок для того ж ключа, або існують прив'язки, що перекриваються – на кожному із цих чотирьох рівнів **tkinter** вибирає "найбільш близьку" з доступних прив'язок.

Приклад: Якщо створено екземпляр прив'язки для події `<Key>` і `<Return>`, буде викликана тільки друга прив'язка, якщо ви натиснете клавішу `Enter`.

Якщо додано зв'язування `<Return>` з верхнього рівня віджета, обидві прив'язки будуть викликатися. `Tkinter` спочатку викликає краще зв'язування на рівні екземпляра, потім краще зв'язування на рівні верхнього рівня вікна, потім краще зв'язування на рівні класу .

Формати методів зв'язування

Кожний віджет `widget` має наступні методи, пов'язані з подіями.

Метод **bind**

widget.bind(*event_name*,*callback*)

Метод **bind_all**

widget.bind_all(*event_name*,*callaback*)

Метод **unbind**

widget.unbind(event_name)

Видаляє всі функції `callback` для події *`event_name`* віджета *`widget`*

Метод **unbind_all**

widget.unbind_all(event_name)

Видаляє всі зворотні виклики (`callback`) для `event_name` на будь-який віджет, раніше встановлений шляхом виклику методу `bind_all` на будь-якому віджеті.

Приклад 9.52.

```
from tkinter import *
root = Tk()
prompt='Click any button, or press a key'
L = Label(root, text=prompt, width=len(prompt))
L.pack()
def key(event):
    if event.char == event.keysym:
        msg ='Normal Key %r' % event.char
    elif len(event.char) == 1:
        msg ='Punctuation Key %r (%r)' % (event.keysym, event.char)
    else:
        msg ='Special Key %r' % event.keysym
    L.config(text=msg)
#---------------------------------------------------------------
L.bind_all('<Key>', key)
```

```
def do_mouse(eventname):
   def mouse_binding(event):
      msg = 'Mouse event %s' % eventname
      L.config(text=msg)
   L.bind_all('<%s>'%eventname, mouse_binding)
#------------------------------------------------------------
for i in range(1,4):
   do_mouse('Button-%s'%i)
   do_mouse('ButtonRelease-%s'%i)
   do_mouse('Double-Button-%s'%i)
root.mainloop( )
```

Інші **callback-***подібні методи*

Для довільного віджета `widget` можна застосувати наступні `callback`-подібні методи.

Метод **after**

***widget*.after(*ms*,*callbback*,**args*)**

Метод **after_cancel**

widget.after_cancel(*id*)

Скасовує таймер, визначений через ID.

Метод **after_idle**

widget.after_idle(callback,*args)

Приклад 9.53.

```
import tkinter
import time
curtime = ''
clock = tkinter.Label()
clock.pack()
def tick():
    global curtime
```

```
    newtime = time.strftime('%H:%M:%S')
    if newtime != curtime:
        curtime = newtime
        clock.config(text=curtime)
    clock.after(200, tick)

tick()
clock.mainloop()
```

Опитування

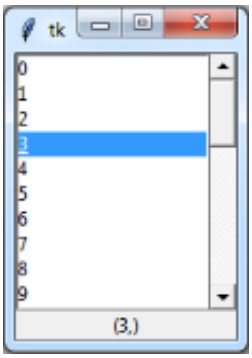
Вид опитування, яке дозволяє реалізувати метод `after`, є дуже важливою технологією бібліотеки `tkinter`. Деякі віджети `tkinter` не мають зворотних викликів, які могли б дозволити довідатися про дії користувачів з ними. Тому якщо ви прагнете відслідковувати такі дії в режимі реального часу, то постійне опитування, наразі, залишається єдиним варіантом.

Наприклад, застосуємо опитування, установлене з використанням `after` для відстеження виборів у віджеті `Listbox` у режимі реального часу:

Приклад 9.54.

```
from tkinter import *
root=Tk()
root.title("Опитування")
F1 = Frame(  )
F2 = Frame()
s = Scrollbar(F1)
L = Listbox(F1,font = "Arial 20", selectmode=MULTIPLE)
L['yscrollcommand'] = s.set
s['command'] = L.yview
#--------------------------
for i in range(30):
    L.insert(END, str(i))
```

```
lab =Label(F2,font = "Arial 20")
def poll(  ):
    lab.after(200, poll)
    sel = L.curselection( )
    lab.config(text=str(sel))
#----------------------------
s.pack(side=RIGHT, fill=Y)
L.pack(side=LEFT, fill=Y)
F1.pack(side=TOP)
lab.pack(  )
F2.pack(side=TOP)
poll()
mainloop()
```



Протоколи

На додачу до прив'язки подій `tkinter` також підтримує механізм, називаний **оброблювачі протоколів.**

Термін протокол відноситься до взаємодії між програмою і менеджером вікон. Найбільш широко використовуваний протокол називається WM_DELETE_WINDOW, і використовується для визначення того, що відбувається, коли користувач явно закриває вікно за допомогою диспетчера вікон.

Можна використовувати метод протоколу, щоб установити оброблювач для цього протоколу (віджет повинен бути `root` або `Toplevel` віджет):

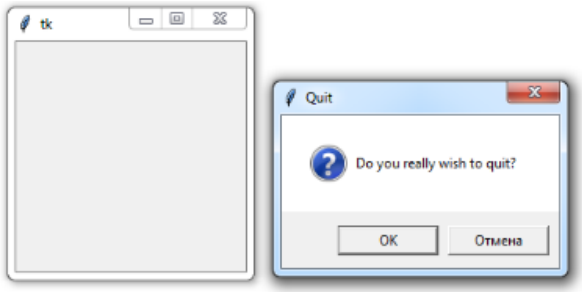
```
widget.protocol("WM_DELETE_WINDOW", handler)
```

handler – оброблювач

Після того, як встановлено свій власний оброблювач, tkinter більше не буде автоматично закривати вікно. Замість цього, можна, наприклад, відобразити вікно повідомлення із запитом користувачеві, якщо поточні дані повинні бути збережені, а в деяких випадках, просто ігнорувати запит. Щоб закрити вікно із цього оброблювача, просто викликати метод **destroy** вікна.

Приклад 9.55. Перехоплення події **destroy**

```
from tkinter import *
from tkinter import messagebox
def callback():
    if messagebox.askokcancel("Quit", "Do you really wish to
quit?"):
        root.destroy()
root = Tk()
root.protocol("WM_DELETE_WINDOW", callback)
root.mainloop()
```



Зверніть увагу, що, навіть якщо ви не зареєстрували оброблювач WM_DELETE_WINDOW для Toplevel вікна, вікно само по собі буде знищено. Проте, з міркувань надійності краще завжди реєструвати оброблювач самостійно:

Приклад 9.56. Toplevel і root з перехопленням події destroy

```
from tkinter import *
from tkinter import messagebox
def callback1():
    if messagebox.askokcancel("toplevel", "Do you really
```

```
wish to quit?"):
        top.destroy()
def callback2():
    if messagebox.askokcancel("root", "Do you really wish to
quit?"):
        root.destroy()
root = Tk()
root.title("Root window")
root.protocol("WM_DELETE_WINDOW", callback2)
top = Toplevel(root, height=500, width=500)
top.protocol("WM_DELETE_WINDOW", callback1)
top.title("Toplevel window with own handler")
top.mainloop()
```

Віджет `Listbox`

Listbox – це віджет, який являє собою список, з елементів якого користувач може вибирати один або кілька пунктів. Має додаткову властивість selectmode,

selectmode = SINGLE, дозволяє користувачеві вибрати тільки один елемент списку

selectmode = EXTENDED, дозволяє користувачеві вибрати будь-яку кількість елементів списку

insert(index, *elements)- вставка елементів

delete(start,stop) – видалення елементів

get(first, last=None)- зчитування елементів

Приклад 9.57.

```
from tkinter import *
root=Tk()
root.title("СПИСОК")
en =Entry(root, width=20,bd=3, font = ("Arial",20))
en.insert(END,"КПІ")
lab_ext=Label(text= "EXTENDED",font = ("Arial",14))
```

```
lbox_ext=Listbox(root,height=5,width=15, font=("Arial",16))
lab_singl=Label(text= "SINGLE",font = ("Arial",14))
lbox_singl=Listbox(root,height=5,width=15,
font=("Arial",16))
list1=["Київ","Хмельницький","Львів","Одеса"]
list2=["Канберра","Сідней","Мельбурн","Аделаїда"]
for i in list1:
    lbox_ext.insert(END,i)
for i in list2:
    lbox_singl.insert(END,i)
def callback():
    lbox_ext.insert(ACTIVE,en.get())
b = Button(root, text="Insert", width=10, command=callback,
font = ("Arial",20))
def callback1():
    lbox_singl.delete(ACTIVE,END)
b1 = Button(root, text="Delete", width=10,
command=callback1, font = ("Arial",20))
def callback2():
    lab_ext.config(text=lbox_ext.get(ACTIVE))
b2 = Button(root, text="get", width=10, command=callback2,
font = ("Arial",20))
def callback3():
    lbox_singl.insert(ACTIVE,lbox_ext.get(ACTIVE))
b3 = Button(root, text="Move", width=10, command=callback3,
font = ("Arial",20))
en.pack()
b.pack()
lab_ext.pack()
lbox_ext.pack()
```

```
lab_singl.pack()
lbox_singl.pack()
b1.pack()
b2.pack()
b3.pack()
root.mainloop()
```

Віджет **Frame**

Віджет `Frame` (рамка) призначений для організації віджетів усередині вікна.

Приклад 9.58.

```
# encoding: utf-8
from tkinter import *
root=Tk()
frame1=Frame(root,bg='green',bd=5)
frame2=Frame(root,bg='red',bd=5)
button1=Button(frame1,text='Перша кнопка')
button2=Button(frame2,text='Друга кнопка')
frame1.pack()
frame2.pack()
button1.pack()
button2.pack()
root.mainloop()
```

Властивість `bd` відповідає за товщину краю рамки.

Віджет **Checkbutton**

`Checkbutton` – дозволяє відзначити „галочкою“ пункт у вікні. При використанні декількох пунктів потрібно кожному присвоїти свою змінну.

`IntVar()` – спеціальний клас бібліотеки для роботи із цілими числами.

`StringVar()` – спеціальний клас бібліотеки для роботи із рядками.

`variable` - властивість, відповідальна за прикріплення до віджету змінної.

onvalue, offvalue – властивості, які присвоюють прикріпленій до віджету змінній значення, яке залежить від стану (onvalue – при вибраному пункті, offvalue – при невибраному пункті).

Приклад 9.59.

```
from tkinter import *
root=Tk()
var1=IntVar()
var2=StringVar()
def fch1():
    print(var1.get())
def fch2():
    print(var2.get())
check1=Checkbutton(root,text='1 пункт для вибору',
variable=var1,onvalue=10,offvalue=0,command=fch1,
font="Arial 16 bold")
check2=Checkbutton(root,text='2 пункт для вибору',
variable=var2, onvalue="Yes", offvalue="No", command = fch2,
font="Arial 16 bold")
check1.pack()
check2.pack()
root.mainloop()
```

Віджет **Radiobutton**

Віджет Radiobutton виконує функцію, схожу з функцією віджета Checkbutton. Різниця в тому, що у віджеті Radiobutton користувач може вибрати лише один з пунктів. Реалізація цього віджета дещо інша, ніж віджета Checkbutton:

variable – властивість, яка закріплює змінну до Radiobutton.

value – властивість, яка надає значення прикріпленій до віджету змінній.

anchor=W – встановлює положення якоря «захід - West»

Дані встановлюємо та зчитуємо через методи змінної `get()` та `set()`.

Приклад 9.60.

```
from tkinter import *
root=Tk()
var=IntVar()
vars=StringVar()
vars.set("one")
var.set(1)
def func1():
    print(var.get())
def func2():
    print(vars.get())
rbutton1=  Radiobutton(root, text='1',      variable=var,
value=1)
rbutton2=  Radiobutton(root, text='2',      variable=var,
value=2)
rbutton3=  Radiobutton(root, text='3',      variable=var,
value=3)
rbutton1s=Radiobutton(root,text='one',   variable=vars,
value="one")
rbutton2s=Radiobutton(root,text='two',   variable=vars,
value="two")
rbutton3s=Radiobutton(root,text='three',variable=vars,
value="three")
button1=Button(root,text='Чтаємо першу групу',
command=func1)
button2=Button(root,text='Чтаємо другу групу',
command=func2)
rbutton1.pack(anchor=W)
rbutton2.pack(anchor=W)
```

```
rbutton3.pack(anchor=W)
rbutton1s.pack(anchor=W)
rbutton2s.pack(anchor=W)
rbutton3s.pack(anchor=W)
button1.pack(anchor=W)
button2.pack(anchor=W)
root.mainloop()
```

Приклад 9.61.

```
from tkinter import *
root = Tk()
v = IntVar()
v.set(1)
languages = [("Python",1), ("Perl",2), ("Java",3),
("C++",4), ("C",5)]
lb = Label(root, text="""Choose your favourite programming
language:""",
justify = LEFT, padx = 20)
lb.pack()
def ShowChoice():
    print(v.get())
for txt, val in languages:
    rbl=Radiobutton(root, text=txt, padx = 20, variable=v,
command=ShowChoice, value=val)
    rbl.pack(anchor=W)
mainloop()
```

Віджет **Scale**

`Scale` (шкала) – це віджет, що дозволяє вибрати будь-яке значення із заданого діапазону.

Властивості:

- **orient** – як розташована шкала у вікні. Можливі значення: HORIZONTAL, VERTICAL (горизонтально, вертикально).
- **length** – довжина шкали.
- **from_** – з якого значення починається шкала.
- **to** – яким значенням закінчується шкала.
- **tickinterval** – інтервал, через який відображаються мітки шкали.
- **resolution** – крок пересування (мінімальна довжина, на яку можна пересунути движок)

Приклад 9.62.

```
from tkinter import *
root = Tk()
def getV(root):
   a = scale1.get()
   print("Значення", a)
scale1 = Scale(root,orient=HORIZONTAL,length=300, from_=50,
to=80,tickinterval=5,resolution=1)
button1 = Button(root,text="Отримати значення")
scale1.pack()
button1.pack()
button1.bind("<Button-1>",getV)
root.mainloop()
```

Тут використовується спеціальний метод `get()`, який дозволяє зняти з віджета певне значення, і використовується не тільки в `Scale`.

Scrollbar

Цей віджет дає можливість користувачеві "прокрутити" інший віджет (наприклад, listbox) і часто буває корисним.

Використання цього віджета досить нетривіально.

Необхідно зробити дві прив'язки:

command смуги прокручування прив'язуємо до методу **xview/yview** віджета,

а **xscrollcommand/yscrollcommand** віджета прив'язуємо до методу **set** смуги прокручування.

Приклад 9.63.

```
from tkinter import *
root = Tk()
root.title("Скролбар")
scrollbar = Scrollbar(root)
scrollbar.pack(side=RIGHT, fill=Y)
listbox = Listbox(root, font="Arial 16 bold")
listbox.pack()
for i in range(100):
    listbox.insert(END, i)
# attach listbox to scrollbar
listbox.config(yscrollcommand=scrollbar.set)
scrollbar.config(command=listbox.yview)
mainloop()
```

Що таке пакувальник ?

Пакувальник або менеджер геометрії, менеджер розташування.

Пакувальник - це спеціальний механізм, який розміщає (упаковує) віджети на вікні.

В `tkinter` є три пакувальники:

pack, place, grid.

Зверніть увагу, що в одному віджеті можна використовувати тільки один тип упакування, при змішуванні різних типів упакування програма, швидше за все, не буде працювати.

Розглянемо кожний з них.

Пакувальник **pack()**

Менеджер геометрії **pack()** пакує віджети в рядки або стовпці.

Для керування цим менеджером можна використовувати його режим за замовчуванням, або задавати такі опції геометрії:

`fill`, **`expand`** і **`side`**.

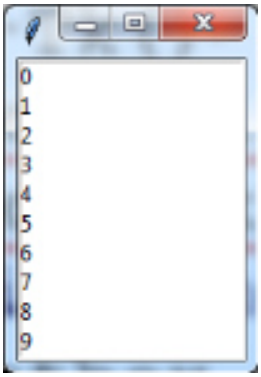
Застосування за замовчуванням.

Розміщуємо віджет усередині контейнера-віджета, і він заповнює весь батьківський віджет.

Приклад: `Listbox` у кореневому вікні:

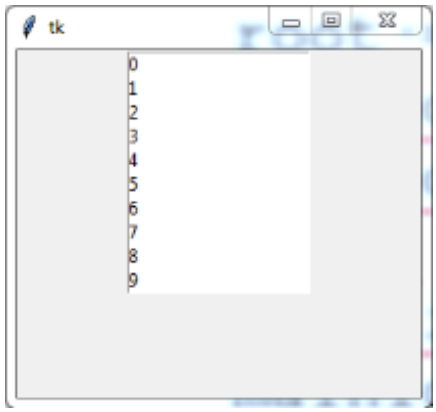
Приклад 9.64. Повне заповнення батьківського віджета.

```
from tkinter import *
root = Tk()
root.title("Packe")
listbox = Listbox(root, font="Arial 16 bold")
listbox.pack()
for i in range(20):
    listbox.insert(END, str(i))
mainloop()
```



Пакування без розтягування

За замовчуванням `Listbox` має розмір 10 елементів. Однак наш `Listbox` містить 20 елементів. Але якщо спробувати показати їх усі за допомогою зміни розмірів вікна, то `tkinter` додасть відступи навколо `Listbox`:



Приклад 9.65. Пакування з розтягуванням.

```
from tkinter import *
root = Tk()
root.title("Гумова верстка")
listbox = Listbox(root, font= "Arial 16 bold")
for i in range(20):
    listbox.insert(END, str(i))
listbox.pack(fill=BOTH, expand=True)
mainloop()
```

Опція **`fill`** указує менеджерові, що віджет прагне заповнити весь простір, призначений для нього.

Опція має допустимі значення:

1) `fill=BOTH` - віджет розширюється як горизонтально, так і вертикально
2) `fill=X` - віджет розширюється тільки горизонтально
3) `fill=Y` - віджет розширюється тільки вертикально.
4) `expand=True`- опція дозволяє розширювати віджет

Упакування віджетів один над одним

Щоб помістити кілька віджетів у стовпці, можна використовувати метод **`pack()`** без будь-яких опцій:

Приклад 9.66. Упакування віджетів один над одним.

```
from tkinter import *
root = Tk()
root.title("Простий pack")
```

```
w = Label(root, text="Червона label",bd=20, bg="red",
fg="white", font="Arial 16 bold")
w.pack()
w = Label(root, text="Зелена label",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack()
w = Label(root, text="Синя label", bd=20, bg="blue",
fg="white", font="Arial 16 bold")
w.pack()
mainloop()
```

Упакування віджетів з горизонтальним розширенням

Приклад 9.67. Розтягування віджетів по ширині батьківського віджета.

```
from tkinter import *
root = Tk()
root.title("Гума по X")
w = Label(root, text="Червона label",bd=20, bg="red",
fg="white", font="Arial 16 bold")
w.pack(fill=X, expand=True)
w = Label(root, text="Зелена label",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack(fill=X, expand=True)
w = Label(root, text="Синя label", bd=20, bg="blue",
fg="white", font="Arial 16 bold")
w.pack(fill=X, expand=True)
mainloop()
```

Упакування віджетів з вертикальним розширенням

Приклад 9.68. Розтягування віджетів по висоті батьківського віджета.

```
from tkinter import *
root = Tk()
root.title("Гума по Y")
```

```
w = Label(root, text="Червона label",bd=20, bg="red",
fg="white", font="Arial 16 bold")
w.pack(fill=Y, expand=True)
w = Label(root, text="Зелена label",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack(fill=Y, expand=True)
w = Label(root, text="Синя label", bd=20, bg="blue",
fg="white", font="Arial 16 bold")
w.pack(fill=Y, expand=True)
mainloop()
```

Упакування віджетів одного поруч із іншим

Для упакування віджетів поруч, використовуємо опцію **`side`.**

Приклад 9.69.

```
from tkinter import *
root = Tk()
root.title("Віджети поряд")
w = Label(root, text="Червона",bd=20, bg="red",
fg="white",font="Arial 16 bold")
w.pack(side=LEFT)
w = Label(root, text="Зелена",bd=20, bg="green",
fg="yellow", font="Arial 16 bold")
w.pack(side=LEFT)
w = Label(root, text="Синя", bd=20, bg="blue", fg="white",
font="Arial 16 bold")
w.pack(side=LEFT)
mainloop()
```

Упакування віджетів з вертикальним розширенням

Якщо необхідно зробити віджети по висоті рівними батьківському, використовуємо додатково опцію `fill=Y`.

Приклад 9.70.

```
from tkinter import *
root = Tk()
root.title("Віджети поряд і гума по Y")
w = Label(root, text="Червона",bd=20, bg="red", fg="white",
font="Arial 16 bold")
w.pack(side=LEFT,fill="y", expand=True)
w = Label(root, text="Зелена",bd=20, bg="green", fg="black",
font="Arial 16 bold")
w.pack(side=LEFT,fill=Y,  expand=True)
w = Label(root, text="Синя",bd=20, bg="blue",
fg="white",font="Arial 16 bold")
w.pack(side=LEFT,fill=Y,  expand=True)
mainloop()
```

Пакування з різними притисканнями у одному вікні

Приклад 9.71.

```
from tkinter import *
root=Tk()
button1 = Button(text="1", font = ("Arial", 20))
button2 = Button(text="2", font = ("Arial", 20))
button3 = Button(text="3", font = ("Arial", 20))
button4 = Button(text="4", font = ("Arial", 20))
button1.pack(side='top')
button2.pack(side='bottom')
button3.pack(side='left')
button4.pack(side='right')
root.mainloop()
```

Пакування з притисканнями та розтягуваннями у вікні¶

```
from tkinter import *
root=Tk()
```

```
root.title("Різні притискання")
button1 = Button(text="Верх", font = ("Arial", 20),
bg="red")
button2 = Button(text="Низ", font = ("Arial", 20), bg="red")
button3 = Button(text="Зліва", font = ("Arial", 20),
bg="blue",fg="white")
button4 = Button(text="Справа", font = ("Arial",
20),bg="blue",fg="white")
button1.pack(side='top', fill=Y, expand=True)
button2.pack(side='bottom', fill=Y, expand=True)
button3.pack(side='left',fill=X, expand=True)
button4.pack(side='right',fill=X, expand=True)
root.mainloop()
```

Огляд опцій пакувальника

anchor – указує точку відліку для урахування розміщення віджета .

За замовчуванням `anchor=CENTER`

Можливі значення опції `anchor`:

1)`anchor=W` – положення «West»

2)`anchor=E` – положення «East»

3)`anchor=N` – положення «North»

4)`anchor=S` – положення «South»

expand (розгортати).

`True` – віджет заповнює вільний простір.

`False` – (за замовчуванням) віджет не розширюється.

Застосування **anchor** показано на прикладі:

Приклад 9.72.

```
from tkinter import *
root=Tk()
button1 = Button(text="1", font = ("Arial", 20))
```

```
button2 = Button(text="2", font = ("Arial", 20))
button3 = Button(text="3", font = ("Arial", 20))
button4 = Button(text="4", font = ("Arial", 20))
button1.pack(anchor=W)
button2.pack(anchor=E)
button3.pack(anchor=N)
button4.pack(anchor=S)
root.mainloop()
```

fill (заповнювати)

Припустимі значення:

X або - 'x' – заповнювати горизонтально,

Y або 'y' – заповнювати вертикально,

BOTH або 'both' – в обох напрямках,

NONE або 'none' – не заповнювати.

ipadx і ipady

Відстань, що вказує, який проміжок повинен залишатися усередині в кожної сторони дочірнього елемента керування.

padx і pady

Відстань, що вказує, який проміжок повинен залишатися ззовні в кожної сторони дочірнього елемента керування.

side (сторона)

Припустимі значення:

LEFT або "left" (ліворуч),

RIGHT або "right" (праворуч),

TOP або "top" (нагорі),

BOTTOM або "bottom" (унизу).

"Внутрішні" елементи керування (дочірні) упаковуються якнайближче до заданої сторони зовнішнього елемента (батька).

Більш докладну інформацію про пакувальника та його опції можна знайти у відповідній документації.

Пакувальник **grid()**

Цей пакувальник є таблицею з комірками, у яких містяться віджети.

Синтаксис: **widget.grid(<аргументи>)**

Аргументи

- **row** – номер рядка, у який поміщаємо віджет.
- **rowspan** – скільки рядків займає віджет
- **column** – номер стовпця, у який поміщаємо віджет.

  За замовчуванням `column = 0` (Самий лівий стовпець)
- **columnspan** – скільки стовпців займає віджет.

Приклад 9.73.

```
from tkinter import *
root = Tk()
label1 = Label(root, text="example",font = ("Arial", 20))
button1 = Button(root, text = '1',font = ("Arial", 20))
button2 = Button(root, text = '2',font = ("Arial", 20))
label1.grid(row=0,column=1,columnspan=3)
button1.grid(row=1,column=0)
button2.grid(row=1,column=4)
root.mainloop()
```

- **padx / pady** – розмір зовнішньої границі (бордюру) по горизонталі й вертикалі.
- **ipadx / ipady** – розмір внутрішньої границі (бордюру) по горизонталі й вертикалі. Різниця між **pad** і **ipad** у тому, що при вказівці **pad** розширюється вільний простір, а при **ipad** розширюється віджет, який поміщаємо.

- **`sticky`** (**"n"**, **"s"**, **"e"**, **"w"** або їх комбінація) – указує, до якої границі "приклеювати" віджет. Дозволяє розширювати віджет у зазначеному напрямку. Границі названі відповідно до сторін горизонту: **"n"** (північ) – верхня границя, **"s"** (південь) – нижня, **"w"** (захід) – ліва, **"e"** (схід) – права.

Контрольні запитання до розділу 9.

1. Яка основна послідовність кроків при створенні GUI?
2. Опишіть порядок завантаження tkinter і створення головного вікна.
3. Який порядок створення віджетів бібліотеки tkinter?
4. Опишіть порядок використання подій та їх обробників.
5. Які базові властивості мають віджети бібліотеки tkinter?
6. Як використовують віджети messagebox?
7. Надайте приклад використання віджетів Entry та Text.
8. Надайте приклад використання Listbox/
9. Які віджети можна застосовувати для керування програмою?
10. Які пакувальники ви знаєте та які переваги вони мають?
11. Використання методу bind для прив’язки подій.
12. Опишіть способи упаковки віджетів у полі вікна.

10. ОБРОБКА ВИКЛЮЧЕНЬ

10.1. Види помилок

10.1.1. Означення виключення

Виключення – це повідомлення інтерпретатора, які виникають у випадку виникнення помилки в програмному коді або при настанні якої-небудь події.

Якщо в коді не передбачена обробка виключення, то виконання програми переривається, і виводиться повідомлення про помилку.

```
a = (1,2,3,4,5)
a[0] = 10
```

Результат роботи:

```
Traceback (most recent call last):
  File "C:/PYTHON/proba.py", line 2, in <module>
    a[0] = 10
TypeError: 'tuple' object does not support item assignment
```

10.1.2. Синтаксичні помилки

Синтаксичні помилки – це помилки в імені оператора або функції, тобто помилки в синтаксисі мови. Як правило, інтерпретатор попередить про наявність помилки, а програма не буде виконуватися зовсім.

Приклад синтаксичної помилки:

```
print("Немає закриваючих лапок!)
File "C:/PYTHON/proba.py", line 1
    print("Немає закриваючих лапок!)
                                     ^
SyntaxError: EOL while scanning string literal
```

10.1.3. Логічні помилки

Логічні помилки – це помилки в логіці програми, які можна виявити тільки за результатами її роботи.

Завдання. Знайти перетин (!) множин X та Y

```
X={1,2,3,4,5}
Y={4,5,6,7,8}
Z=X | Y
print("Z = ",Z)
```

Результат:

```
Z =  {1, 2, 3, 4, 5, 6, 7, 8}
```

Відповідь неправильна, оскільки в програмі існує логічна помилка: неправильно записана логічна операція

Потрібно було записати &, а записано |

10.1.4. Помилки часу виконання

Помилки часу виконання – це помилки, які виникають під час роботи програми. Причиною є події, які не передбачені програмістом.

Класичним прикладом служить ділення на нуль:

Приклад 10.1.

```
def test(x, y):
   return x / y
x=int(input("Ведіть целе число(ділене)"))
y=int(input("Ведіть целе число(дільник)"))
z=test(x,y)
```

```
ZeroDivisionError: division by zero
```

Приклад помилки часу виконання

Приклад 10.2.

```
a=[199,112,-3,4,5,6,7]
b=[12,1,127,-11,34,56]
```

```
def difer1(arr1,arr2):
    return  list(map(lambda i,j: i-j, arr1,arr2))
def difer2(arr1,arr2):
    for i in range(len(arr1)):
        arr1[i]=arr1[i]-arr2[i]
    return  arr1
print(difer1(a,b))
print(difer2(a,b))
```

Результат роботи програми:

[187, 111, -130, 15, -29, -50]

```
Traceback (most recent call last):
File "C:/MYPYTHON/Lecture28/Ex2.py", line 15, in
<module>
    print(difer2(a,b))
  File "C:/MYPYTHON/Lecture28/Ex2.py", line 10, in
difer2
    arr1[i]=arr1[i]-arr2[i]
IndexError: list index out of range
```

10.2. Обробка помилок

10.2.1. Виключення при настанні події

В мові Python виключення виконуються не тільки при помилці, але і як повідомлення про настання яких-небудь подій. Наприклад, метод `index()` виконує виключення **`ValueError`**, якщо шуканий фрагмент не входить у рядок:

Приклад 10.3.

```
a = "Рядок".index("т")
b = [1,3,4]
c= b.pop(5)
print(c, b)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: substring not found
```

Інструкція ***try...except...else...finally***

Для обробки виключень призначена інструкція `try`.

Формат інструкції:

try:
<Блок, у якому перехоплюються виключення>
except [<Виключення1>[**as** <Об'єкт виключення>]]**:**
<Блок, виконуваний при виникненні виключення 1>
[except [<Виключення2>[**as** <Об'єкт виключення2>]]**:**
<Блок, виконуваний при виникненні виключення2>]
[**else:**
<Блок, виконуваний, якщо виключення не виникло>]
[**finally:**
<Блок, виконуваний у будь-якому випадку>]

Найпростіший варіант інструкції **try**…**except**

Інструкції, у яких перехоплюються виключення, повинні бути розташовані усередині блоку **try**.

У блоці **except** у параметрі <Виключення1> вказують клас оброблюваного виключення.

Наприклад, обробити виключення, що виникає при діленні на нуль, можна так, як показано в прикладі.

Приклад 10.4.

```
def devisor(a):
    try:    # Перехоплюємо виключення
        x = 1/a    # Помилка: ділення на 0
    except ZeroDivisionError:#Указуємо клас виключення
        print("Опрацювали ділення на 0")
        x = 0
    return x
print(devisor(0))
Результат виконання :
Опрацювали ділення на 0
    0
```

10.2.2. Інструкція **try**…**except** *у загальному випадку*

Якщо в блоці **try** виникло виключення, то керування передається блоку **except**.

Якщо виключення не відповідає зазначеному класу, керування передається наступному блоку **except**.

Якщо жоден блок except не відповідає виключенню, то виключення «спливе» до обробника більш високого рівня.

Якщо виключення в програмі взагалі ніде не обробляється, воно передається обробнику за замовчуванням, який зупиняє виконання програми й виводить стандартну інформацію про помилку.

Приклад 10.5.

```
arr=[1,2,3,4,5]
def compdev(a,b,c):
    try:    # Опрацьовуємо виключення
        for i in range(c):
            a[i]/=b # Помилка: ділення на 0
    except ZeroDivisionError:
         print("Опрацювання ділення на 0")
         x = 0
    except IndexError:
        print("Неіснуючий індекс")
    return a

print(compdev(arr,0,len(arr)))
print(compdev(arr,2,len(arr)))
print(compdev(arr,2,len(arr)+3))
```

Обробники **try**…**except** *можуть бути вкладеними*

Приклад 10.5. Вкладені обробники.

```
arr=[1,2,3,4,5]
def compdev(a,b,c):
```

```
    try:   # Опрацьовуємо виключення
        try: # Вкладений обробник
            for i in range(c):
                a[i]/=b # Помилка: ділення на 0
        except IndexError:
            print("Неіснуючий індекс")
        print("Вкладений обробник пройшли")
    except ZeroDivisionError:
        print("Опрацювання ділення на 0")
        x = 0
    return a
print(compdev(arr,0,len(arr)))
print(compdev(arr,2,len(arr)))
print(compdev(arr,2,len(arr)+3))
```

Результати роботи прикладу

```
Опрацювання ділення на 0
[1, 2, 3, 4, 5]
Вкладений обробник пройшли
[0.5, 1.0, 1.5, 2.0, 2.5]
Неіснуючий індекс
Вкладений обробник пройшли
[0.25, 0.5, 0.75, 1.0, 1.25]
```

Кілька виключень

В інструкції **except** можна вказати відразу кілька виключень, перелічивши класи через кому усередині круглих дужок.

Приклад 10.6.

```
arr1=[1,2,3,4,5]
def dilen(a,b,c):
    try:
        for i in range(c):
```

```
            a[i] /= b
    except (IndexError, ZeroDivisionError):
        print("Виникла якась помилка")
        return a
print(dilen(arr1,0,len(arr1)))
print(dilen(arr1,2,len(arr1)+3))
Виникла якась помилка
[1, 2, 3, 4, 5]
Виникла якась помилка
[0.5, 1.0, 1.5, 2.0, 2.5]
```

Інформація про оброблене виключення

Одержати інформацію про оброблюване виключення можна через другий параметр в інструкції **except.**

Приклад 10.7.

```
arr1=[1,2,3,4,5]
def dilen(a,b,c):
    try:
            for i in range(c):
                a[i] /= b
        except (NameError, IndexError, ZeroDivisionError) as err:
            print(err.__class__.__name__)
            print(err)
        return a
    print(dilen(arr1,0,len(arr1)))
print(dilen(arr1,2,len(arr1)+3))
```

ZeroDivisionError division by zero [1, 2, 3, 4, 5]	IndexError list index out of range [0.5, 1.0, 1.5, 2.0, 2.5]

Функція **exc_info()**

Одержання інформації про виключення

Функція знаходиться в модулі **sys** та повертає кортеж із трьох елементів:

1. тип виключення, (Type)
2. значення (Value)
3. об'єкт з трасувальною інформацією. (Trace)

Перетворити ці значення в зручний для читання вигляд дозволяє модуль traceback.

Приклад використання функції **exc_info()** і модуля **traceback** наведений у прикладі.

Приклад 10.8.

```
import sys, traceback
try:
    x = 1 / 0
except ZeroDivisionError:
    Type, Value, Trace = sys.exc_info()
    print("Type: ", Type)
    print("Value:", Value)
    print("Trace:", Trace)
```

```
Результат виконання фрагменту:
Type:  <class 'ZeroDivisionError'>
Value: division by zero
Trace: <traceback object at 0x002F7CD8>
```

Продовження прикладу

```
print("\n", "print exception()".center(40, "-"))
traceback.print_exception(Type, Value, Trace, limit=5, file=sys.stdout)
```

```
----------print exception()------------
Traceback (most recent call last):
  File "C:/MY/first.py", line 3, in <module>
    x = 1 / 0
```

```
ZeroDivisionError: division by zero
print("\n", "print_tb()".center(40, "-" ))
traceback.print_tb(Trace, limit=1, file=sys.stdout)
      ----------print_tb()----------
  File "C:/MY/first.py", line 3, in <module>
x = 1 / 0
print("\n", "format_exception()". center ( 40, "-") )
print(traceback.format_exception(Type, Value, Trace, limit=5))
      ----------format_exception()----------
['Traceback (most recent call last):\n', '  File "C:/MY/first.py", line 3, in <module>\n
x = 1 / 0\n', 'ZeroDivisionError: division by zero\n']
print("\n", "format_exception_only()".center(40, "-"))
print(traceback.format_exception_only(Type, Value))
    --------format_exception_only()--------
['ZeroDivisionError: division by zero\n']
```

Блок `else` – інструкції усередині цього блоку будуть виконані тільки при відсутності помилок.

Блок `finally` – виконуються завершальні дії незалежно від того, чи виникло виключення, чи ні.

Приклад 10.10 Застосування **except…else…finally**

```
a=int(input("Введіть дільник: "))
try:
    x = 10 / a    # Помилка ділення на 0
except ZeroDivisionError:
    print ( "Ділення на 0")
else: print("Виключення не відбулося")
finally: print("Працює завжди")
```

Результат виконання:

Введіть дільник: 0 Ділення на 0 Працює завжди	Введіть дільник: 1 Виключення не відбулося Працює завжди

Інструкція **try…finally (без блоку except)**

Інструкції усередині блоку **finally** будуть виконані, але виключення не буде оброблено.

Воно продовжить «спливання» до обробника більш високого рівня.

Якщо обробник користувача відсутній, то керування передається обробнику за замовчуванням, який перериває виконання програми й виводить повідомлення про помилку.

Приклад 10.11.

```
try:
    x= int(input("Введіть число:"))
finally: print("Вводимо дані")
```

Результат виконання:

```
Введіть число:проба
Вводимо дані
Traceback (most recent call last):
  File "C:/MY/first.py", line 2, in <module>
    x= int(input("Введіть число:"))
ValueError: invalid literal for int() with base 10:
'проба'
```

Ще одна перевірка вводу

Приклад 10.12.

```
print("Введіть слово 'stop' для отримання результату")
suma = 0
while True:
    x = input("Введіть число: ")
    if x == "stop":
        break # Вихід з циклу
    try:
        x=float(x) # Перетворюємо рядок на число
    except ValueError:
```

```
            print("Необхідно ввести число!")
        else:
            suma += x
print("Сума чисел дорівнює:",round(suma,2))
```

Процес введення значень і одержання результату має такий вигляд (значення, введені користувачем, виділені напівжирним шрифтом):

```
Введіть слово 'stop' для отримання результату
Введіть число: 10
Введіть число: -5
Введіть число: 34,67
Необхідно ввести число!
Введіть число: 34.45
Введіть число: 56.23
Введіть число: stop
Сума чисел дорівнює: 95.68
```

*Інструкція **with...as***

Інструкція призначена для обгортання блоку інструкцій менеджером контексту.

Формат інструкції:

with <**Класс**>[**as**<Результуюча змінна >]**:**

<Код, де відбувається перехоплення виключення>

Послідовність дій:

1. Виконується метод класу**__enter__**

2. Якщо конструкція **with** включає слово **as**, то метод **__enter__**, повертає значення в результуючу змінну.

3. Виконується <... перехоплення виключення>

4. Викликається метод класу **__exit__**.

5.В метод **__exit__** передаються параметри виключення **Type Value Trace**, якщо воно сталося, або у всіх аргументах значення **None**, якщо виключення не було.

Метод **__exit__**

Формат методу:

__exit__(self, <**Type**>, <**Value**>, <**Trace**>)

Значення, доступні через останні три параметри, еквівалентні значенням, що повертаються функцією **exc_info ()** з модуля **sys**.

Якщо виключення оброблене, метод повинен повернути значення **True**, а якщо ні, то – **False**.

Якщо метод повертає **False**, то виключення передається вищому обробнику.

Розглянемо послідовність виконання протоколу на прикладі.

Приклад 10.13. Протокол менеджерів контексту**.**

```
class MyClass:
    def __enter__(self):
        print("Спрацював метод __enter_()")
        return self
    def __exit__ (self, Type, Value, Trace):
        print ("Спрацював метод _exit_() ")
        print("Type:",Type,", Value:", Value, ", Trace:",Trace)
        if Type is None:# Якщо виключення не виникло
            print("Виключення не виникло")
            return False
        else:
            print("Value =", Value)
            return True
print("Код до місця пошуку виключення:")
with MyClass()as obj:
    print("Код, в якому шукаємо виключення ")
```

Код до місця пошуку виключення:

```
Спрацював метод __enter_()
Код, в якому шукаємо виключення
Спрацював метод _exit_()
Type: None , Value: None , Trace: None
Виключення не виникло
```

```
class MyClass:
    def __enter__(self):
        print("Спрацював метод __enter_()")
        return self
    def __exit__ (self, Type, Value, Trace):
        print ("Спрацював метод _exit_() ")
        if Type is None:# Якщо виключення не виникло
            print("Виключення не виникло")
            return False
        else: # Якщо виникло виключення
            print("Value =", Value)
            return True
print("Код за наявності виключення:")
with MyClass() as obj:
    print("Код, в якому шукаємо виключення ")
    x=12+"a"
```

```
Код за наявності виключення:
Спрацював метод __enter_()
Код, в якому шукаємо виключення
Спрацював метод _exit_()
Value = unsupported operand type(s) for +: 'int' and 'str'
```

Підтримка **with**...**as** *за замовчуванням*

Деякі вбудовані об'єкти підтримують протокол за замовчуванням – наприклад, файли.

Якщо в інструкції **with** зазначена функція **open(),** то після виконання інструкцій усередині блоку файл автоматично буде закритий. Приклад використання інструкції **with:**

Приклад 10.14.

```
with open ("test. txt", "a", encoding="utf-8") as f:
    f.write("Рядок\n") #записуємо рядок у кінець файлу
```

У цьому прикладі файл **test.txt** відкривається на дозапис у кінець файлу.

Після виконання функції **open()** змінній **f** буде присвоєно посилання на об'єкт файлу.

За допомогою цієї змінної ми можемо працювати з файлом всередині тіла інструкції **with.**

Після виходу із блоку незалежно від наявності виключення файл буде закритий.

10.2.3. Виключення користувача

Для виконання виключень користувача призначено дві інструкції: **raise** і **assert**.

Інструкція raise виконує задане виключення. Вона має кілька варіантів формату:

```
raise<Екземпляр класу>
raise <Назва класу>
raise <Екземп. або назва класу> from <Об'єкт виключ>
raise
```

У першому варіанті формату інструкції raise Вказується екземпляр класу в якому виникло виключення.

При створенні екземпляра можна передати дані конструктору класу.

Ці дані будуть доступні через другий параметр в інструкції **except**.

```
raise <екземпляр класу користувача>
```

Приклад 10.15.

Створюємо свій клас обробки помилки для нашого екземпляру класу

```
class MyError(Exception):
    def __init__(self, value):
        self.msg = value
    def __str__(self):
        return self.msg
# Опрацювання виключення користувача
err= MyError("Опис виключення1")
try:
    raise err
except MyError as my_err:
    print(my_err) #Викликаємо метод __str__ ()
    print(my_err.msg) #Доступ до атрибуту класу
```

```
Опис виключення1
Опис виключення1
```

Пустий клас, який спадкує **Exception**

Клас **Exception** містить усі необхідні методи для виводу повідомлення про помилку. Тому в більшості випадків достатньо створити порожній клас, який успадковує клас **Exception**:

Приклад 10.16.

```
class MyError(Exception): pass
err= MyError("Опис виключення")
try:
    raise err
except MyError as my_err:
    print (my_err)
```

Результат:

```
Опис виключення
```

```
raise <Назва класу>
```

У другому варіанті формату інструкції **raise** у першому параметрі задають клас, а не екземпляр.

Приклад 10.17.

```
import sys
try:
    raise FileNotFoundError
# Еквівалентно: raise ValueError()
except OSError:
    print("Повідомлення про помилку")
    Type, Value, Trace = sys.exc_info()
    print(Type)
```

Результат роботи:

```
Повідомлення про помилку
<class 'FileNotFoundError'>
```

Приклад 10.18.

Приклад виконання вбудованого виключення `ValueError` з власним описом:

```
raise ValueError ("Мій власний опис виключення")
Traceback (most recent call last):
  File "C:/MY/first.py", line 1, in <module>
    raise ValueError ("Мій власний опис виключення")
ValueError: Мій власний опис виключення
```

Приклад обробки цього виключення:

```
import sys
try:
    raise FileExistsError("Опис виключення")
except OSError as msg:
    print(msg)
    Type, Value, Trace = sys.exc_info()
```

```
    print(Type, Value)
```

Результат виконання:

Опис виключення

<class 'FileExistsError'> Опис виключення

raise <Екземпляр або назва класу> from <Об'єкт>

У третьому варіанті формату інструкції **raise** у першому параметрі задають екземпляр класу або просто назва класу, а в другому параметрі вказують об'єкт виключення.

При обробці вкладених виключень ці дані використовуються для виводу інформації не тільки про останнє виключення, але й про перше виключення.

В наступному прикладі ми одержали інформацію не тільки по виключенню **ValueError**, але й по виключенню **ZeroDivisionError**. При відсутності інструкції `from` інформація зберігається неявним чином.

Приклад 10.19. Послідовність помилок з **from**

```
try:  #(інструкція from відсутня)
    x = 10 / 0
except Exception as err:
    raise ValueError() from err
```

Результат виконання:

```
Traceback (most recent call last):
  File "C:/MY/first.py", line 2, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

The above exception was the direct cause of the
following exception:

Traceback (most recent call last):
  File "C:/MY/first.py", line 4, in <module>
    raise ValueError() from err
```

```
try:            #(інструкція from відсутня)
    x = 1 / 0
except Exception as err:
    raise ValueError()
```

Результат виконання:

```
Traceback (most recent call last):
  File "C:/MY/first.py", line 2, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "C:/MY/first.py", line 4, in <module>
    raise ValueError()
ValueError
```

raise

Інструкція **raise** дозволяє повторно виконати останнє виключення й зазвичай застосовується в коді, наступному за інструкцією **except.**

Приклад 10.20.

```
class MyError(Exception): pass
try:
    raise MyError("Повідомлення про помилку")
except MyError as err:
    print (err)
    raise
```

Результат виконання:

```
Повідомлення про помилку
```

```
Traceback (most recent call last):
     File "C:/PYTHON/first.py", line 3, in <module>
       raise MyError ("Повідомлення про помилку")
   __main__. MyError: Повідомлення про помилку
```

Інструкція **assert**

Інструкція `assert` виконує виключення `AssertionError`, якщо логічний вираз повертає значення `False`. Інструкція має наступний формат:

```
assert <Логічний вираз> [, <Дані>]
```

Інструкція `assert` еквівалентна наступному коду:

```
if not <логічний вираз>:
        raise AssertionError()
```

Приклад 10.21.

```
assert True
print("Нічого не відбулося")
assert False
print("Виникла помилка")
```

Результат:

```
Нічого не відбулося
Tracback.....
AssertionError
```

Приклад використання інструкції **assert**:

Приклад 10.22.

```
def fchecker (a):
   try:
       x = a*2
       assert x >= 2, "Помилка: x<2"
       print("Код успішно виконано")
   except AssertionError as err:
       print(err)
```

```
fchecker(10)
```

Результат:

```
Код успішно виконано
```

```
fchecker(-10)
```

Результат:

```
Помилка: x<2
```

Контролні запитання до розділу 10.

1. Дайте визначення виключення.
2. Синтаксичні помилки та способи їх визначення.
3. Логічні помилки та помилки часу виконання. Способи їх запобігання.
4. Застосування варіантів інструкцій try… except
5. Надайте приклад застосування функції exc_info().
6. Яке призначення інструкції with…as?
7. Опишіть умови застосування та дію методу __exit__().
8. Інструкція raise та її застосування для створення виключень користувача.
9. Яке виключення формує інструкція assert?
10. Опишіть спосіб створення класу, який спадкує Exception?

Розділ 11. Регулярні вирази

Що таке регулярні вирази ?

Просто кажучи, регулярний вираз - це **послідовність символів, яка використовується для пошуку і заміни тексту в рядку або файлі.**

Їх підтримує безліч мов загального призначення: Python, Perl, Java і т.і. Тому вивчення регулярних виразів рано чи пізно стане в нагоді.

У мові Python використовувати регулярні вирази дозволяє модуль **re**.

Перед використанням функцій із цього модуля необхідно підключити модуль за допомогою інструкції:

```
import re
```

Найчастіше регулярні вирази використовуються для:

пошуку в рядку;

розбиття рядка на підрядки;

заміни частини рядка.

11.1. Основні функції модуля re

- **`re.match()`** – ця функція шукає відповідність заданому шаблону на початку рядка.
- **`re.search()`** – шукає відповідність заданому шаблону у всьому рядку, але повертає тільки першу знайдену відповідність.
- **`re.findall()`** – повертає список всіх знайдених відповідностей.
- **`re.split()`** – розділяє рядок на підрядки за заданим шаблоном
- **`re.sub()`** – шукає шаблон у рядку та замінює його на заданий підрядок
- **`re.compile()`** – дозволяє зібрати регулярний вираз в окремий об'єкт, який може бути використаний для пошуку. Це також дозволяє позбутись переписування одного і того ж виразу

11.1.1. Функції re.match() *та* re.search()

Результат пошуку функціями re.match() та re.search()

Якщо пошук не досяг успіху, то функції повертають None.

Якщо шуканий підрядок знайдено,то функції повертають об’єкт типу

<class 're.Match'>.

Цей об’єкт має наступні методи:

Метод **group ()** для виводу знайденого підрядка

Метод **start ()** для виводу початкової позиції знайденого рядка.

Метод **end()** для виводу кінцевої позиції знайденого рядка.

Метод **span()** для виводу кортежу з початкової і кінцевої позиції знайденого рядка.

Використовуємо «r» перед рядком шаблону, щоб показати, що це «сирий» рядок в Python.

Приклад 11.1. Застосування функції **re.match()**

```
import re
result = re.match(r'Мій', 'Мій перший пошук')
result1 = re.match(r'перший', 'Мій перший пошук')
if result:
   print(type(result))
   print ("Знайдено:",result.group(0))
   print ("Початковий індекс:",result.start())
   print ("Кінцевий індекс:",result.end())
if result1: print (result1.group(0))
else: print ("Результат:",result1)
```

Результати роботи

```
<class 're.Match'>
Знайдено: Мій
Початковий індекс: 0
Кінцевий індекс: 3
Результат: None
```

Приклад 11.2. Застосування функції **re.search()**

```
import re
result = re.search(r'другий', 'Мій другий пошук')
```

```
result1 = re.search(r'перший', 'Мій другий пошук')
if result:
   print(type(result))
   print ("Знайдено:",result.group(0))
   print ("Початковий індекс:",result.start())
   print ("Кінцевий індекс:",result.end())
if result1: print (result1.group(0))
else: print ("Результат:",result1)
```

Результати роботи

```
<class 're.Match'>
Знайдено: другий
Початковий індекс: 4
Кінцевий індекс: 10
Результат: None
```

11.1.2. Функція `re.findall()`

Приклад 11.3. Застосування функції `re.findall()`

Якщо пошук не досяг успіху, то функція повертає пустий список. Якщо шукані підрядки знайдено, то функція повертає об’єкт типу <class 'list'>.

```
import re
result = re.findall(r'ш', 'Я шукаю перший раз')
result1 = re.findall(r'перший', 'Мій другий пошук')
if result:
   print(type(result))
   print ("Знайдено:",result)
else: print ("Результат:",result)
if result1: print (result1)
else: print ("Результат:",result1)
```

Результат: <class 'list'>

Знайдено: ['ш', 'ш']

Результат: []

*Що таке шаблон (pattern)***?**

Ми розглядали шаблон як послідовність символів.

В Python можливо задати також шаблон на простій мові, яка дозволяє шукати підрядок у рядку за певними правилами.

Використовуючи цю мову, вказуємо правила для довільної кількості можливих рядків, які ми хочемо перевірити.

Ці рядки можуть містити англійські фрази, адреси електронної пошти, команди, все що завгодно.

За допомогою шаблона можна одержати відповідь на такі питання:

«Чи відповідає цей рядок шаблону?»,

«Чи збігається шаблон з цим рядком частково?».

11.1.3. Основні метасимволи

Метасимволи – це спеціальні символи, які вказують на те, що задана деяка певна умова. Вони можуть впливати на інші частини регулярного виразу, змінюючи їх значення.

Основні метасимволи: **. ^ $ * + ? {} [ ] \ | ( )**

[та **]** – використовуються для визначення класу символів, що є набором символів, з якими шукаємо збіг.

Символи можуть бути перераховані окремо, або у вигляді деякого діапазону символів, позначеного першим і останнім символом, розділених знаком '-'.

Приклад 11.4:

[abc]- шаблон відповідає будь-якому символові: a, b, c;

[a-c] -аналогічно можна задати діапазон.

[a-z] – всі малі латинські літери.

Приклади класів в [ ]

[09] – відповідає числу 0 або 9;

[0-9] – відповідає будь-якому числу від 0 до 9;

[абв] – відповідає буквам «а», «б» і «в»;

[а-яіґ] – відповідає будь-якій букві від «а» до «я»;

[АБВ] – відповідає буквам «А», «Б» і «В»;

[А-ЯІҐ] – відповідає будь-якій букві від «А» до «Я»;

[а-яА-ЯіІґҐ] – відповідає будь-якій кириличній букві ;

ПРИМІТКА

Букви «**і**» та «ґ» не входять у діапазон [а–я], а букви «**І**» та «**Ґ**» – у діапазон [А-Я].

Значення в дужках інвертується, якщо після першої дужки вставити символ ^. У такий спосіб можна вказати символи, яких не повинно бути на цьому місці в рядку:

```
[^09]- не цифра 0 або 9;
[^0-9]- не цифра від 0 до 9;
[^а-яА-ЯіІґҐa-zA-Z]- не буква.
```

Поведінка метасимволів в **[ ]**

Інші метасимволи, крім ^ на початку, не активні всередині класів.

Приклад 11.5.

[akm $] - буде відповідати будь-якому з символів 'a', 'k', 'm' або '$'.

Знак '$' це зазвичай метасимвол (як видно зі списку символів вище), але всередині класу символів він позбавляється своєї особливої природи.

Для того, щоб знаходити відповідність символам поза класом, на початку класу додається символ '^'. Наприклад, вираз [^5] відповідає будь-якому символу, крім '5'.

Метасимвол **\'**

1.Метасимвол \ використовується для екранування метасимволов, щоб їх можна було використовувати в шаблонах.

Приклад 11.6.

Нехай потрібно знайти відповідність **[** або ****.

Для того щоб позбавити їх своєї особливої ролі метасимволів, перед ними треба поставити зворотний слеш: \[або \\.

2.Деякі зі спеціальних послідовностей, що починаються з '\' є спеціальними наборами символів, які корисні для скорочення регулярного виразу. Розглянемо такі спеціальні послідовності.

*11.1.4. Спеціальні послідовності з метасимволом *

\d - відповідає будь-якій цифрі; еквівалент класу [0-9].

\D - відповідає будь-якому нечисловому символу;

еквівалент класу [^0-9].

\s- відповідає будь-якому символу **whitespace**;

еквівалент [\t\n\r\f\v].

\S - відповідає будь-якому не-whitespace символу;

еквівалент [^\t\n\r\f\v].

\w - відповідає будь-якій букві або цифрі;

еквівалент [a-zA-Z0-9_].

\W - навпаки; еквівалент [^a-zA-Z0-9_].

\b – прив'язка до початку слова

(початком слова вважається пробіл або будь-який символ, що не є буквою, цифрою або знаком підкреслення);

\B- прив'язка до позиції, що не є початком слова.

Ці послідовності можуть бути включені в клас символів.

Приклад 11.7.

[\s,.] - клас, який буде відповідати будь-якому whitespace-символу або комі (,) або точці (.).

Метасимвол **'.'**

Метасимвол '.' відповідає всім символам, крім символу нового рядка \n, але є альтернативний режим (re.DOTALL), при якому буде включатися і '.' .

Метасимвол '.' використовується там, де необхідно порівняти будь-які символи.

Метасимвол прив'язки

^ – прив'язка до початку рядка або підрядка.

$ – прив'язка до кінця рядка або підрядка.

\A – прив'язка до початку рядка.

\Z – прив'язка до кінця рядка.

Метасимвол альтернативи `I`

Метасимвол I дозволяє зробити вибір між альтернативними значеннями.

Вираз **`n|m`** відповідає одному із символів: **`n`** або **`m`**.

Метасимволи входжень

`{n}` – `n` входжень символу в рядок.

Приклад 11.8.

Шаблон **`r"[0-9]{2}$"`** відповідає двом входженням будь-якої цифри;

`{n,m}` – не менше `n` і не більше `m` входжень символу в рядок. Числа вказуються через кому без пробілу. Наприклад, шаблон `r"[0-9]{2,4}$"` відповідає від двох до чотирьох входжень будь-якої цифри;

Метасимволи входжень (продовження)

`*` – нуль або більше число входжень символу в рядок. Еквівалентно комбінації**`{0,};`**

`+` – одне або більше число входжень символу в рядок. Еквівалентно комбінації **`{1,}`**;

`?` – жодного або одне входження символу в рядок. Еквівалентно комбінації `{0,1}`.

Синтаксис регулярних виразів

Створити відкомпільований шаблон регулярного виразу дозволяє функція `compile()`.

Функція має наступний формат:

<Об'єкт пошуку>=**`re.compile`**(<Регулярний вираз>[,<Модифікатор>])

У параметрі <Модифікатор> можуть бути зазначені прапори компіляції.

Прапори компіляції дозволяють змінювати деякі аспекти того, як працюють регулярні вирази.

Прапори доступні в модулі під двома іменами: довгим, таким як IGNORECASE і коротким, в однобуквеній формі, таким як I. Кілька прапорів можуть бути задані у формі двійкового АБО; наприклад re.I | re.M встановлює прапори I і M.

11.1.5. Прапори компіляції

ASCII, A вибирає тільки ASCII-символи

DOTALL, S відповідність така ж як '.', тобто з будь-яким символом, але при включенні цього прапора, в розгляд додається і символ нового рядка.

IGNORECASE, I відповідність без урахування регістру;

Приклад 11.9.

[A-Z] буде також відповідати і малим буквам, так що Spam буде відповідати Spam, spam, spAM і так далі.

LOCALE, L - робить \ w, \ W, \ b, \ B залежними від локалізації. Наприклад, якщо ви працюєте з текстом українською, і хочете написати \w + для того, щоб знаходити слова, але \w шукає тільки символи з множини [A-Яа-я] і не буде шукати 'ґ'. Якщо система налаштована правильно і задано прапор re.L , 'ґ' також буде розглядатися як буква.

MULTILINE, M – зазвичай ^ шукає відповідність тільки на початку рядка, а $ тільки в кінці безпосередньо перед символом нового рядка \n (якщо такі є). Якщо прапор M вказано, ^ порівняння відбувається у всіх рядках, тобто і на початку, і відразу ж після кожного символу нового рядка. Аналогічно для $.

UNICODE, U - робить \ w, \ W, \ b, \ B, \ d, \ D, \ s, \ S відповідними таблиці Unicode.

VERBOSE, X включає багатослівні (докладні) регулярні вирази, які можуть бути організовані більш ясно і зрозуміло. Якщо вказаний цей прапор, пробіли в рядку регулярного виразу ігнорується, крім випадків, коли вони є в класі символів або їм передує неекранований зворотний слеш; дозволяє поміщати в регулярні вирази коментарі, що починаються з '#', які будуть ігноруватися движком.

11.2. Основні методи об’єкта пошуку

<**Об’єкт пошуку**>.match() – цей метод шукає відповідність заданому шаблону на початку рядка.

<**Об’єкт пошуку**>.search() – шукає відповідність заданому шаблону у всьому рядку, але повертає тільки першу знайдену відповідність.

<**Об’єкт пошуку**>.findall() – повертає список всіх знайдених відповідностей.

<**Об’єкт пошуку**>.fullmatch()– виконує перевірку, чи відповідає переданий рядок регулярному виразу цілком. Підтримка цього методу з'явилася в Python.

<**Об’єкт пошуку**>.finditer()–аналогічно findall(), але повертає ітератор, а не список

Приклад 11.10. Пошук єдиної відповідності і пошук одної (або більше) відповідності для кожної букви

```
import re
p=re.compile(r"[а-яіґ]", re.I | re.U)
p1=re.compile(r"[а-яіґ]+", re.I | re.U)
s = p.search("АБВГДЕ")
s1 = p1.search("АБВГДЕ")
s2 = p1.search("абвгдАБВГДЕ")
if s:print(s.span())#повертає кортеж (start(),end())
print(s.group(0))
if s1: print(s1.span())
```

```
print(s1.group(0))
if s2: print(s2.span())
print(s2.group(0))
```

Результат:

(0, 1)

A

(0, 6)

АБВГДЕ

(0, 11)

абвгдАБВГДЕ

Приклад 11.11. Перевірка на розпізнавання символу переходу на новий рядок з різними прапорами

```
import re
def check(a):
    if a:print(repr(a.group()))
         print(a.span())
    else: print(a)
p = re.compile(r".$")
check(p.search("\n"))
p1 = re.compile(r".$", re.M)
check(p1.search("\n"))
p2 = re.compile(r".$", re.S)
check(p2.search("\n"))
```

Результат:

```
None
None
'\n'
(0, 1)
```

11.2.1. Прив'язка до початку й кінця рядка

Прив'язку до початку й кінця рядка слід використовувати, якщо рядок повинен повністю відповідати регулярному виразу. Наприклад, для перевірки, чи містить рядок число.

Приклад 11.12. В рядку мають бути тільки цифри.

```
import re
p = re.compile (r"^[0-9]+$") #одне або більше входження
print("Знайдено" if p.search("1234567890") else "Hi")
print("Знайдено" if p.search("q1234567890") else "Hi")
print("Знайдено" if p.search("1234567890q") else "Hi")
print("Знайдено" if p.search("1234567q890") else "Hi")
print("Знайдено" if p.search("") else "Hi")
```

Результат:

```
Знайдено
Hi  Hi Hi Hi
```

Якщо забрати прив'язку до початку й кінця рядка, то будь-який рядок, що містить хоча б одну цифру, буде розпізнаний як `Число`

Приклад 11.13. Пошук без прив'язки до початку й кінця рядка

```
import re # Підключаємо модуль
p = re.compile(r"[0-9]+", re.S)
a=p.search("Рядок245")
if a:
    print("Число в рядку")
    print(a.span())
    print(a.group(0))
else: print("Немає числа в рядку")
```

Результат: Число в рядку

```
(5, 8)
245
```

Крім того, можна вказати прив'язку тільки до кінця рядка

Приклад 11.14. Пошук з прив’язкою тільки до кінця рядка**.**

```
import re # Підключаємо модуль
p = re.compile(r"[0-9]+$", re.S)
```

```
if p.search("Рядок245"):
    print("Є число наприкінці рядка")
else:
    print("Немає числа наприкінці рядка")
```

Результат: Є число наприкінці рядка

Приклад 11.15. Пошук з прив'язкою тільки до початку рядка

```
import re # Підключаємо модуль
p = re.compile(r"^[0-9]+", re.S)
if p.search("Рядок245"):
    print("Є число на початку рядка")
 else:
    print("Немає числа на початку рядка")
```

Результат: Немає числа на початку рядка

Приклад 11.16. Прив'язка до початку рядка за допомогою \b

```
import re
p = re.compile(r"\bpython")
print("Знайдено" if p.search("python") else "Hi")
print("Знайдено" if p.search("pythonware") else "Hi")
print("Знайдено" if p.search("thon") else "Hi")
print("Знайдено" if p.search("pyth") else "Hi")
```

Результат: Знайдено Знайдено Hi Hi

```
#Прив'язка до початку і кінця рядка за допомогою \b
import re
p = re.compile(r"\bpython\b")
print("Знайдено" if p.search("python") else "Hi")
print("Знайдено" if p.search("pythonware") else "Hi")
print("Знайдено" if p.search("thon") else "Hi")
print("Знайдено" if p.search("pyth") else "Hi")
```

Результат: Знайдено Hi Hi Hi

Приклад 11.17. В рядку мають бути тільки цифри або пустий рядок

```
import re
p = re.compile (r"^[0-9]*$")#нуль або більше входжень
print("Знайдено" if p.search("") else "Hi")
print("Знайдено" if p.search("12334567890") else "Hi")
print("Знайдено" if p.search("w12334567890") else "Hi")
print("Знайдено" if p.search("123w3457890") else "Hi")
print("Знайдено" if p.search("wwwww") else "Hi")
```

Результат:

```
Знайдено
Знайдено
Hi
Hi
Hi
```

Приклад 11.18. В рядку має бути тільки одна цифра або пустий рядок

```
import re
#жодного або одне входження
p = re.compile (r"^[0-9]?$")
    print("Знайдено" if p.search("") else "Hi")
print("Знайдено" if p.search("1") else "Hi")
print("Знайдено" if p.search("6") else "Hi")
print("Знайдено" if p.search("w") else "Hi")
print("Знайдено" if p.search("w1")else "Hi")
```

Результат:

```
Знайдено
Знайдено
Знайдено
Hi
Hi
```

Приклад 11.19. Перевірка формату дати

```
import re # Підключаємо модуль
d = "29,12.2017"  # Замість крапки вказана кома
p = re.compile(r"^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$")
if p.search(d): print("Дата введена правильно")
else: print("Дата введена неправильно")
p = re.compile(r"^[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]$")
if p.search(d): print("Дата введена правильно")
else:   print("Дата введена неправильно")
p = re.compile (r"^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$")
if p. search(d):   print("Дата введена правильно")
else:  print("Дата введена неправильно")
```

Результат:

```
Дата введена правильно {крапка-метасимвол}
Дата введена неправильно
Дата введена неправильно
```

Приклад 11.20. Вплив модифікаторів (прапорів)

```
import re
# Крапка не відповідає \n
p = re.compile(r"^.+$")
print(p.findall("str1\nstr2\nstr3"))
# Тепер крапка відповідає \n
p = re.compile(r"^.+$",re.S)
print(p.findall("str1\nstr2\nstr3"))
# Багаторядковий режим
p = re.compile(r"^.+$",re.M)
print(p.findall("str1\nstr2\nstr3"))
```

Результат

```
[]
['str1\nstr2\nstr3']
```

```
['str1', 'str2', 'str3']
```

Метасимвол I дозволяє зробити вибір між альтернативними значеннями. Вираз `n|m` відповідає одному із символів: `n` або `m`.

Приклад 11.21.

```
import re
p = re.compile(r"червон((а)|(е))")
print("Знайдено" if p.search("червона") else "Ні")
print("Знайдено" if p.search("червоне") else "Ні")
print("Знайдено" if p.search("червоний") else "Ні")
print(p.findall("червона"))
```

Результат:

```
Знайдено
Знайдено
Ні
[('а', 'а', '')]
```

11.2.2. Пошук `findall()` *по шаблону*

Усі метасимволи є «жадібними». При пошуку відповідності шукається найдовший підрядок, відповідно до шаблону, і не враховуються більш короткі відповідності. Розглянемо це на прикладі й одержимо вміст усіх тегів , разом з тегами:

Приклад 11.22.

```
import re
#* - 0 або більше
s = "<b>Text1</b> Text2\n <b>Text3</b>"
p = re.compile(r"<b>.*</b>", re.S)
print(p.findall(s))
```

Результат:

```
['<b>Text1</b> Text2\n <b>Text3</b>']
```

Даний рядок починається на і закінчується на

Обмеження пошуку `findall()` *по шаблону*

Замість бажаного результату ми одержали повністю рядок. Щоб обмежити «жадібність», необхідно після квантифікатора вказати символ **?**

Приклад 11.23. Обмежена «жадібність» модифікатора.

```
import re
#* - 0 або більше
#? - жодного або одне
s = "<b>Text1\n</b>Text2<b>Text3</b>"
p = re.compile(r"<b>.*?</b>", re.S)
print(p.findall(s))
```

Результат:

```
['<b>Text1\n</b>', '<b>Text3</b>']
```

Цей код вивів те, що ми шукали.

Одержати текст без тегів

Якщо необхідно одержати вміст без тегів, то потрібний фрагмент всередині шаблону слід розмістити всередині круглих дужок.

Приклад 11.24.

```
import re
#* - 0 або більше
#? - жодного або одне
s = "<b>Text1\n</b>Text2<b>Text3</b>"
p = re.compile(r"<b>(.*?)</b>", re.S)
print(p.findall(s))
```

Результат: `['Text1\n', 'Text3']`

Угрупування фрагментів усередині шаблону

Для угруповання фрагментів усередині шаблону будемо користуватися круглими дужками.

У цьому випадку не потрібно, щоб фрагмент запам'ятовувався й був доступний у результатах пошуку, але щоб уникнути захоплення фрагмента.

Для уникнення захоплення слід після відкриваючої круглої дужки розмістити символи ? :

Приклад 11.25.

```
s = "Учні та учителі в школі"
```

Шаблон пошуку: **`([а-я]+((ителі)|(ні)))`**

Зовнішня дужка захоплює елементи «**Учні», «учителі»**

Внутрішні дужки захоплюють елементи «**ителі», «ні»**

Потрібно виключити захоплення цих фрагментів.

Приклад 11.26. *Обмеження захоплення фрагмента*

```
import re
#* - 0 або більше
#? - жодного або одне
s = "Учні та  учителі в школі"
p = re.compile(r"([а-я]+((ителі)|(ні)))", re.I)
print(p.findall(s))
p1 = re.compile(r"([а-я]+(?:(?:ителі)|(?:ні)))", re.I)
print(p1.findall(s))
```

Результат:

```
[('Учні','ні','','ні'),('учителі','ителі','ителі','')]
['Учні', 'учителі']
```

Пояснення до прикладу

У першому прикладі ми одержали список із двома елементами. Кожний елемент списку є кортежем, що містить чотири елементи.

Усі ці елементи відповідають фрагментам, укладеним у шаблоні в круглі дужки. Перший елемент кортежу містить фрагмент, розташований у перших круглих дужках, другий – у других круглих дужках і т. д. Три останні елементи кортежу є зайвими.

Щоб вони не виводилися в результатах, ми додали символи ?: після кожної відкриваючої круглої дужки. У результаті список складається тільки із фрагментів, що повністю відповідають регулярному виразу.

11.2.3. Зворотне посилання на фрагмент

До знайденого фрагмента в круглих дужках усередині шаблону можна звернутися за допомогою механізму зворотних посилань. Для цього порядковий номер круглих дужок у шаблоні вказують після слеша – наприклад, так: \1. Нумерація дужок усередині шаблону починається з 1. Для прикладу одержимо текст між однаковими парними тегами.

Приклад 11.27.

```
import re
s = "<b>Один</b> Два <I>Три</I> <p>Чотири</p>"
p = re.compile(r"<([a-z]+)>(.*?)</\1>", re.S | re.I)
print(p.findall(s))
```

Результат:

```
[('b', 'Один'), ('I', 'Три'), ('p', 'Чотири')]
```

Використання іменованих фрагментів

Фрагментам усередині круглих дужок можна дати імена. Для цього після відкриваючої круглої дужки слід указати комбінацію символів `?P<name>`. Як приклад разберемо e-mail на складові частини:

Приклад 11.28. Іменовані фрагменти

```
import re
email = "test@ukr.net"
p = re.compile(r"(?P<name>[a-z0-9]+)@(?P<host>[a-z0-9]+\.[a-z]{2,6})", re.I )
z=p.search(email)
print(z. group ("name"))
print(z.group("host"))
```

Результат:

```
test
ukr.net
```

Звернення до іменованих фрагментів усередині шаблону

Щоб всередині шаблону звернутися до іменованих фрагментів, використовується наступний синтаксис: (?P=name). Для прикладу одержимо текст між однаковими парними тегами.

Приклад 11.29. Доступ до іменованих фрагментів усередині шаблону:

```
import re
s = "<b>Text1</b>Text2<I>Text3</I>"
p = re.compile(r"<(?P<tag>[a-z]+)>(.*?)</(?P=tag)>", re.S | re.I)
print(p.findall(s))
```

Результат:

```
[('b', 'Text1'), ('I', 'Text3')]
```

Можливі конструкції усередині круглих дужок

(?ailmsux) - дозволяє встановити опції регулярного виразу.

Букви a, i, l, m, s, u і x мають таке ж призначення, що й однойменні модифікатори у функції compile ();

a-тільки ASCII-символи,

i–ignorecase

l–locale

m–multiline

s–будь-який символ, включаючи \n,

u–unicode.

x–допустимі коментарі.

(?#...)- коментар. Текст всередині круглих дужок ігнорується;

Приклад 11.30. Коментар

```
import re
s = "Мій текст"
p = re.compile(r"[а-я,і]+(?#Список слів)",re.S|re.I)
print(p.findall(s))
```

(?=...)– позитивний перегляд вперед.

Приклад 11.31. Виведемо всі слова, після яких розташована кома:

```
import re
s = "Перша_кома, Друга_кома, Без_коми"
p = re.compile(r"\w+(?=[,])", re.S | re.I)
print(p.findall(s))
```

Результат:

```
['Перша_кома', 'Друга_кома']
```

`(?!...)` – негативний перегляд уперед.

Приклад 11.32. Виведемо всі слова, після яких немає коми:

```
import re
s = "text1, text2, text3 text4"
p = re.compile(r"[a-z]+[0-9](?![,])",re.S | re.I)
print(p.findall(s))
```

Результат:

```
['text3', 'text4']
```

`(?<=...)` – позитивний перегляд назад.

Приклад 11.33. Виведемо всі слова, перед якими розташована кома з пробілом:

```
import re
s = "text1, text2, text3 text4"
p = re.compile(r"(?<=[,][ ])[a-z]+[0-9]", re.S | re.I)
print(p.findall(s))
```

Результат:

```
['text2', 'text3']
```

`(?<!...)` - негативний перегляд назад.

Приклад 11.34. Виведемо всі слова, перед якими розташований пробіл, але перед пробілом немає коми:

```
import re
s = "text1, text2, text3 text4"
```

```
p = re.compile(r"(?<![,])([ ][a-z]+[0-9])", re.S | re.I)
print(p.findall(s))
```

Результат: `[' text4']`

Приклад послідовності з дефісами

Необхідно одержати всі слова, розташовані після дефіса, причому перед дефісом і після слів повинні слідувати «пропускові» символи:

Приклад 11.35.

```
import re
s = "-word1 -word2 -word3 -word4 -word5"
p=re.compile(r"\s\-([a-z0-9]+)\s",re.S | re.I)
print(p.findall(s))
```

Результат:

```
['word2', 'word4']
```

Як видно із прикладу, ми одержали тільки два слова замість п'яти. Перше й останнє слова не потрапили в результат, тому що розташовані на початку й наприкінці рядка.

Щоб ці слова потрапили в результат, необхідно додати альтернативний вибір `(^ | \s)` – для початку рядка і `( \s | $)` – для кінця рядка. Щоб знайдені вирази всередині круглих дужок не потрапили в результат, слід додати символи `?:` після відкриваючої дужки:

Приклад 11.36.

```
import re
s = "-word1 -word2 -word3 -word4 -word5"
p = re.compile(r"(?:^|\s)\-([a-z0-9]+)(?:\s|$)", re.S | re.I)
print(p.findall(s))
```

Результат:

```
['word1', 'word3', 'word5']
```

Перше й останнє слова успішно потрапили в результат. Чому ж слова word2 і word4 не потрапили в список збігів – адже перед дефісом є пробіл і після слова є пробіл?

Щоб зрозуміти причину, розглянемо пошук по кроках. Перше слово успішно потрапляє в результат, тому що перед дефісом розташований початок рядка, і після слова є пробіл. Після пошуку покажчик переміщається, і рядок для подальшого пошуку набуде наступного вигляду:

"-word1 <Покажчик>-word2 –word3 -word4 -word5"

Зверніть увагу на те, що перед фрагментом -word2 більше немає пробілу, і дефіс не розташований на початку рядка. Тому наступним збігом буде слово word3, і покажчик знову буде переміщений:

"-word1 -word2 –word3 <Покажчик>-word4 -word5"

Знову перед фрагментом -word4 немає пробілу, і дефіс не розташований на початку рядка. Тому наступним збігом буде слово word5, і пошук буде завершений. Таким чином, слова word2 і word4 не попадають у результат, оскільки пробіл до фрагмента вже був використаний у попередньому пошуку.

Щоб цього уникнути, слід скористатися позитивним переглядом уперед (?=…):

Приклад 11.37.

```
import re
s = "-word1 -word2 -word3 -word4 -word5"
p = re.compile(r"(?:^|\s)\-([a-z0-9]+)(?=\s|$)", re.S | re.I)
print(p.findall(s))
```

Результат:

['word1', 'word2', 'word3', 'word4', 'word5']

У цьому прикладі ми замінили фрагмент (?: \s | $) на (?=\s | $). Тому всі слова успішно потрапили в список збігів.

11.2.4. Огляд методів та функцій пошуку

Формат функції re.match() модуля re:

re.match (<Шаблон>, <Рядок>[, <Модифікатор>])

У параметрі <Шаблон> вказується рядок з регулярним виразом або скомпільований регулярний вираз.

У параметрі **<**Рядок**>** задаємо рядок, у якому відбувається пошук.

У параметрі <Модифікатор> можна вказати прапори, використовувані у функції compile().

Якщо відповідність знайдена, то повертається <**об'єкт пошуку**>, а якщо ні, то повертається значення None.

Приклад 11.38.

```
import re
def check(a):
    if a: print("Знайдено")
    else: print("Ні")
p1 = r"[0-9]+"
check(re.match(p1, "str123"))
check(re .match(p1, "123str"))
p2 = re.compile(r"[0-9]+")
check(re.match(p2, "123str"))
```

Результат роботи:

```
Знайдено
Знайдено
```

Формат методу **<**об'єкт пошуку**>**.match модуля re:

<об'єкт пошуку>.match(<Рядок>[,<Початкова позиція>[,<Кінцева позиція>]])

Якщо відповідність знайдена, то повертається об'єкт Match, а якщо ні, то повертається значення None.

Приклад 11.39.

```
import re
def check(a):
    if a: print("Знайдено")
    else: print("Ні")
tamp = re.compile(r"[0-9]+")
check(tamp.match("str123"))
```

```
check(tamp.match("str123",3))
check(tamp.match("123str"))
```

Результат роботи:

```
Ні
Знайдено
Знайдено
```

Формат функції `re.search()` модуля `re`:

re.search(<Шаблон>, <Рядок>[,<Модифікатор>))

У параметрі **<**Шаблон**>** вказується рядок з регулярним виразом або скомпільований регулярний вираз.

У параметрі **<**Модифікатор**>** можна вказати прапори для використання у функції `compile()` .

У параметрі <Рядок> задаємо рядок, у якому відбувається пошук.

Якщо відповідність знайдена, то повертається об'єкт `Match`, а якщо ні, то повертається значення `None`.

Приклад 11.40.

```
import re
def check(a):
   if a: print("Знайдено")
   else: print("Ні")
p1 = r"[0-9]+"
check(re.search(p1, "str123"))
p2 = re.compile(r"[0-9]+")
check(re.search(p2, "str123"))
check(re.search(p2, "123str"))
```

Результат роботи:

```
Знайдено
Знайдено
Знайдено
```

Формат методу <об’єкт пошуку>.**search** модуля re:

<об’єкт пошуку>.search(<Рядок>[,<Початкова позиція>[,<Кінцева позиція>]])

search() – перевіряє відповідність із будь-якою частиною рядка. Якщо знайдено, то повертається об'єкт Match, а якщо ні, то повертається значення None.

Приклад 11.41.

```
import re
def check(a):
   if a: print("Знайдено", end=' ')
   else: print("Ні", end=' ')
tamp = re.compile(r"[0-9]+")
check(tamp.search("str123"))
check(tamp.search("123str",3))
check(tamp.search("123str"))
```

Результат роботи:

Знайдено Ні Знайдено

Формат функції re. **Fullmatch()** модуля re:

re.fullmatch(<Шаблон>,<Рядок>[,<Модифікатор>])

У параметрі <Шаблон> вказується рядок з регулярним виразом або скомпільований регулярний вираз.

У параметрі **<**Модифікатор**>** можна вказати прапори, використовувані у функції compile() .

У параметрі **<**Рядок**>** задаємо рядок, у якому відбувається пошук.

Якщо рядок повністю збігається з шаблоном, повертається об’єкт Match, а якщо ні, то повертається значення None.

Приклад 11.42.

```
import re
def check(a):
   if a: print("Знайдено")
   else: print("Ні")
```

```
p = "[Pp]ython"
check(re.fullmatch (p,"Python"))
check(re.fullmatch (p,"python"))
check(re.search(p, "py"))
check(re.search(p, "thon"))
```

Результат роботи:

```
Знайдено
Знайдено
Ні
Ні
```

Формат методу <об'єкт пошуку>.fullmach модуля re:

<об'єкт пошуку>.fullmach(<Рядок>[,<Початкова позиція>[,<Кінцева позиція>]])

fullmatch()– виконує перевірку, чи відповідає переданий рядок регулярному виразу цілком.

Приклад 11.43.

```
import re
def check(a):
   if a: print("Знайдено", end=' ')
   else: print("Ні", end=' ')
tamp = re.compile("[Pp]ython")
check(tamp.fullmatch("Python"))
check(tamp.fullmatch("python"))
check(tamp.fullmatch("py"))
check(tamp.fullmatch("nothpy"))
check(tamp.fullmatch("Pythonware",0,6))
```

Результат роботи: Знайдено Знайдено Ні Ні Знайдено

Приклад застосування регулярних виразів

Приклад 11.44. Додавання невизначеної кількості чисел

```
import re
print("Введіть слово 'stop' для отримання результату")
```

```
suma = 0
p = re.compile(r"^[-]?[0-9]+$", re.S)
while True:
   x = input("Введіть число: ")
   if x == "stop":
      break  # Вихід з циклу
   if not p.search(x):
      print("Необхідно ввести число, а не рядок!")
      continue # Переходимо на наступну ітерацію циклу
   x = int(x) # Перетворюємо рядок на число
   suma += x
print("Сума чисел дорівнює:", suma)
```

11.2.5. Атрибути та методи об'єкта **Match**

Об'єкт `Match`, що повертається методами (функціями) `match()` і `search()`, має наступні властивості й методи:

`<об'єкт пошуку>.re.`**groups -** кількість груп у шаблоні;

`<об'єкт пошуку>.re.`**groupindex -** словник з назвами груп і їх номерами;

`<об'єкт пошуку>.re.`**pattern -** початковий рядок з регулярним виразом;

`<об'єкт пошуку>.re.`**flags -** комбінація прапорів, заданих при створенні регулярного виразу у функции `compile()`, і прапорів, зазначених у самому регулярному виразі, у конструкції `(?ailmsux)`;

`<об'єкт пошуку>.`**string** – значення параметра **<**`Рядок`**>** у методах (функціях) `match()` і `search ()`;

`<об'єкт пошуку>.`**pos** – значення параметра **<**`Початкова позиція`**>** у методах `match()` і `search()` ;

<об'єкт пошуку>.**endpos** – значення параметра <Кінцева позиція> у методах match() і search() ;

<об'єкт пошуку>.**lastindex** – повертає номер останньої групи або значення None, якщо пошук завершився невдачею;

<об'єкт пошуку>.**lastgroup** – повертає назва останньої групи або значення None, якщо ця група не має ім'я, або пошук завершився невдачею.

Приклад 11.45.

```
import re
p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
m = p.search("123456string 67890text")
print(m)
print(m.re.groups, m.re.groupindex)
print(m.re.pattern)
print(m.re.flags)
print(m.string)
print(m.lastindex, m.lastgroup)
print(m.pos, m.endpos)
```

Результат роботи:

```
<re.Match object; span=(0, 12), match='123456string'>
2 {'num': 1, 'str': 2}
(?P<num>[0-9]+)(?P<str>[a-z]+)
32
123456string 67890text
2 str
0 22
```

<об'єкт пошуку>.**group([<id1 або name1> [, ... , <idn або namen>]])**

повертає фрагменти, які відповідають шаблону.

Якщо параметр не заданий або зазначене значення 0, повертається фрагмент, повністю відповідний до шаблону.

Якщо зазначений номер або назва групи, повертається фрагмент, що збігається із цією групою.

Через кому можна вказати декілька номерів або назв груп – у цьому випадку повертається кортеж, який містить фрагменти, що відповідають групам.

Якщо немає групи із зазначеним номером чи назвою, то виконується виключення **`IndexError`**.

Приклад 11.46.

```
import re
p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
m = p.search("123456string 67890text")
print("Повна відповідність:",m.group(),m.group(0))
print("Доступ по індексу:",m.group(1), m.group(2))
print("Доступ по назві:",m.group('num'), m.group('str'))
print("Декілька параметрів:",m.group(1,2), m.group('num','str'))
```

Результат роботи:

```
Повна відповідність: 123456string 123456string
Доступ по індексу: 123456 string
Доступ по назві: 123456 string
Декілька параметрів: ('123456', 'string') ('123456', 'string')
```

`<об'єкт пошуку>`**`.groupdict([<Значення за замовчуванням>)`** – повертає словник, що містить значення іменованих груп. За допомогою необов'язкового параметра можна вказати значення, яке буде виводитися замість значення `None` для груп, що не мають збігів:

Приклад 11.47.

```
import re
p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
```

```
m = p.search("123456")
print(m.groupdict())
print(m.groupdict(""))
```

Результат роботи:

```
{'num': '123456', 'str': None}
{'num': '123456', 'str': ''}
```

<об’єкт пошуку>.groups([<Значення за замовчуванням>])– повертає кортеж, що містить значення всіх груп. За допомогою необов'язкового параметра можна вказати значення, яке буде виводитись замість значення `None` для груп, що не мають збігів:

Приклад 11.48.

```
import re
p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
m = p.search("123456")
print(m.groups())
print(m.groups(""))
```

Результат роботи:

```
('123456', None)
('123456', '')
```

<об’єкт пошуку>**.start([<Номер або назва групи>])** – повертає індекс початку фрагмента, який відповідає заданій групі. Якщо параметр не зазначений, то фрагментом є повна відповідність з шаблоном. Якщо відповідності немає, то повертається значення -1;

<об’єкт пошуку>**.end([<Номер або назва групи>])** – повертає індекс кінця фрагмента, відповідного заданій групі. Якщо параметр не зазначений, то фрагментом є повна відповідність з шаблоном. Якщо відповідності немає, то повертається значення -1;

<об’єкт пошуку>**.span([<Номер або назва групи>])** – повертає кортеж, що містить початковий і кінцевий індекси фрагмента,

відповідного до заданої групи. Якщо параметр не зазначений, то фрагментом є повна відповідність з шаблоном. Якщо відповідності немає, то повертається значення (-1, -1) .

Приклад 11.49.

```
import re
p = re.compile(r"(?P<num>[0-9]+)(?P<str>[ a-z]+)")
s = "str123456str"
m = p.search(s)
print(m.start(), m.end(), m.span())
print(m.start(1), m.end(1), m.start("num"), m.end("num"))
print(m.start(2), m.end(2), m.start("str"), m.end("str"))
print(m.span(1), m.span("num"), m.span(2), m. span("str"))
print(s[m.start(1):m.end(1)], s[m.start(2):m.end(2)])
```

Результат роботи:

```
3 12 (3, 12)
3 9 3 9
9 12 9 12
(3, 9) (3, 9) (9, 12) (9, 12)
123456 str
```

<об'єкт пошуку>**.expand(<Шаблон>)** – виконує заміну в рядку.

Усередині зазначеного шаблону можна використовувати зворотні посилання: **\номер групи** й **\g<номер групи>** і **\g<назва групи>**.

Приклад 11.50. Поміняємо два теги місцями:

```
import re
p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
m = p. search("<br><hr>")
print(m.expand(r"<\2><\1>"))
print(m.expand(r"<\g<2>><\g<1>>"))
print(m.expand(r"<\g<tag2>><\g<tag1>>"))
```

Результат роботи:

<hr>

<hr>

<hr>

Використання функції **`search()`**

Перевіримо на відповідність шаблону введену користувачем адресу електронної пошти:

Приклад 11.51.

```
import re
email = input("Введіть e-mail: ")
pe = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pe, re.I | re.S)
m = p.search(email)
if not m: print("E-mail не відповідає шаблону")
else:
   print("E-mail", m.group(0), "відповідає шаблону")
   print ("ящик:", m.group(1), "домен:", m.group(2))
```

Результат виконання:

```
Введіть e-mail: user@ukr.net
E-mail user@ukr.net відповідає шаблону
ящик: user домен: ukr.net
```

Пошук всіх збігів з шаблоном

Формат функції re.**`findall`** модуля re:

`re.findall``(<Шаблон>,<Рядок>[,<Модифікатор>])`

Повертається список із фрагментами, у протилежному випадку повертається порожній список.

Якщо всередині шаблону є більше, ніж одна група, то кожний елемент списку буде кортежем, а не рядком.

У параметрі **<**`Шаблон`**>** вказується рядок з регулярним вираженням або скомпільоване регулярне вираження.

У параметрі **<**`Модифікатор`**>** можна вказати прапори, які використовуються в функції `compile()`.

У параметрі `<Рядок>` задаємо рядок, у якому відбувається пошук.

Приклад 11.52.

```
import re
re.findall(r"[0-9]+", "1 2 3 4 5 6")
p = re.compile(r"[0-9]+")
print(re.findall(p, "1 2 3 4 5 6"))
```

Результат виконання:

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

Формат методу <об'єкт пошуку>. **findall** модуля re:

<об'єкт пошуку>. findall(<Рядок>[,<Початкова позиція>[,<Кінцева позиція>]])

Якщо відповідності знайдені, повертається список із фрагментами, у протилежному випадку повертається порожній список.

Якщо всередині шаблону є більше, ніж одна група, то кожний елемент списку буде кортежем, а не рядком.

Приклад 11.53.

```
import re
p1 = re.compile(r"[0-9]+")
print(p1.findall("2012, 2013, 2014, 2015, 2016"))
p2 = re.compile(r"[a-z]+")
print(p2.findall("2012, 2013, 2014, 2015, 2016"))
t = r"[0-9]{3}-[0-9]{2}-[0-9]{2}"
p = re.compile(t)
print(p.findall("322-55-98"))
print(p.findall("322-55-98, 678-56-12"))
```

Результат виконання:

```
['2012', '2013', '2014', '2015', '2016']
[]
['322-55-98']
['322-55-98', '678-56-12']
```

Формат функції re.**finditer** модуля re:

re.finditer(<Шаблон>,<Рядок>[,<Модифікатор>])

Функція finditer() аналогічна функції findall(), але повертає ітератор, а не список. На кожній ітерації циклу повертається об'єкт Match.

У параметрі **<**Шаблон**>** вказується рядок з регулярним виразом або скомпільований регулярний вираз.

У параметрі **<**Модифікатор**>** можна вказати прапори, використовувані в функції compile().

У параметрі <Рядок> задаємо рядок, у якому відбувається пошук.

Приклад 11.54. Одержимо вміст між тегами:

```
import re
p = re.compile(r"<b>(.+?)</b>", re.I | re.S)
s = "<b>Text1</b>Text2<b>Text3</b>"
for m in re.finditer(p, s):
    print(m.group(1))
```

Результат виконання:

```
Text1
Text3
```

Формат методу **<**об'єкт пошуку**>**.**finditer** модуля re:

<об'єкт пошуку>.finditer(<Рядок>[,<Початкова позиція>[,<Кінцева позиція>]])

Метод finditer() аналогічний методу findall(), але повертає ітератор, а не список. На кожній ітерації циклу повертається об'єкт Match.

Приклад 11.55.

```
import re
p = re.compile(r"[0-9]+")
for m in p.finditer("2012, 2013, 2014, 2015, 2016"):
  print(m.group(0), "start:", m.start(), "end:", m.end())
```

Результат виконання:

```
2012 start: 0 end: 4
2013 start: 6 end: 10
2014 start: 12 end: 16
2015 start: 18 end: 22
2016 start: 24 end: 28
```

11.2.6. Заміна в рядку. Метод **<об'єкт пошуку>.sub**

Метод `sub()` шукає всі збіги із шаблоном і заміняє їхнім зазначеним значенням. Якщо збіги не знайдені, повертається початковий рядок. Метод має наступний формат:

sub(<фрагмент або посилання на функцію>, <Рядок для заміни>[,<Максимальна кількість замін>])

Усередині нового фрагмента можна використовувати зворотні посилання **\номер групи, \g<номер групи>** і **\g<назва групи>.**

Приклад 11.56. Поміняємо два теги місцями:

```
import re
p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
print(p.sub(r"<\2><\1>", "<br><hr>")) # \номер
print(p.sub(r"<\g<2>><\g<1>>", "<br><hr>"))# \g<номер>
print(p.sub(r"<\g<tag2>><\g<tag1>>", "<br><hr>")) # \g<назва>
```

Результат виконання:

```
<hr><br>
<hr><br>
<hr><br>
```

Як перший параметр можна вказати посилання на функцію.

Приклад 11.57.

```
import re
def repl(m):
```

```
    """ Функція для заміни. m – об'єкт Match """
    x = int(m.group(0))
    x += 10
    return "{0}".format(x)
p = re.compile(r"[0-9]+")
# Заміняємо всі входження
print(p.sub(repl, "2012, 2013, 2014, 2015"))
# Заміняємо тільки перші два входження
print(p.sub(repl, "2012, 2013, 2014, 2015", 2))
```

Результат виконання:

```
2022, 2023, 2024, 2025
2022, 2023, 2014, 2015
```

Функція **`re.sub()`**

re.sub(<Шаблон>, <фрагмент або посилання на функцію>, <Рядок для заміни>[,<Максимальна кількість замін> [, flags =0])

Приклад 11.58. Поміняємо два теги місцями, а також змінимо регістр букв:

```
import re
def repl(m) :
    tag1 = m.group("tag1").upper()
    tag2 = m.group("tag2").upper()
    return "<{0}><{1}>".format(tag2, tag1)
p = r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>"
print(re.sub(p, repl, "<br><hr>"))
```

Результат виконання:

<HR>

Метод **`<об'єкт пошуку>.subn()`**

Метод `subn()` аналогічний методу `sub()`, але повертає не рядок, а кортеж із двох елементів: зміненого рядка й кількості зроблених замін.

Метод має наступний формат:

<об'єкт пошуку>.subn(<фрагмент або посилання на функцію>, <Рядок для заміни>[, <максимальна кількість замін>])

Замінимо всі числа в рядку на 0:

Приклад 11.59.

```
import re
p = re.compile(r"[0-9]+")
print(p.subn("0", "2014, 2015, 2016, 2017"))
```

Результат виконання:

```
('0, 0, 0, 0', 4)
```

Функція **re.subn()**

Формат функції:

re.subn(<Шаблон>, <Новий фрагмент або посилання на функцію>,<Рядок для заміни>[,<Максимальна кількість замін> [, flags=0]])

Приклад 11.60.

```
import re
p = r"201[5]"
print(re.subn(p, "2012", "2014, 2015, 2016, 2017"))
```

Результат виконання: ('2014, 2012, 2016, 2017', 1)

Метод **<об'єкт пошуку>.expand()**

Для виконання замін також можна використовувати метод expand(), підтримуваний объектом Match.

Формат методу: **<об'єкт пошуку>.expand(<Шаблон>)**

Усередині зазначеного шаблону можна використовувати зворотні посилання: **\номер групи, \g<номер групи>** і **\g<назва групи>.**

Приклад 11.61

```
import re
p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
```

```
m = p.search("<br><hr>")
print(m.expand(r"<\2><\1>"))  #\номер
print(m.expand(r"<\g<2>><\g<1>>"))# \g<номер>
print(m.expand(r"<\g<tag2>><\g<tag1>>")) # \g<назва>
```

Результат виконання:

```
<hr><br>    <hr><br>     <hr><br>
```

Метод **<об'єкт пошуку>.split()**

Метод `split()` розбиває рядок по шаблону й повертає список підрядків. Його формат:

<об'єкт пошуку>.split(<Початковий рядок>[, <Ліміт>])

Якщо в другому параметрі <Ліміт> задане число, то в списку опиниться зазначена кількість підрядків.

Якщо підрядків більше від зазначеної кількості, то список буде містити ще один елемент – із залишком рядка.

Приклад 11.62.

```
import re
p = re.compile(r"[\s,.]+")
print(p.split("word1, word2\nword3\r\nword4.word5"))
print(p.split("wordl, word2\nword3\r\nword4.word5", 2))
```

Результат виконання:

```
['word1', 'word2', 'word3', 'word4', 'word5']
['wordl', 'word2', 'word3\r\nword4.word5']
```

Якщо роздільник у рядку не знайдений, то список буде складатись тільки з одного елемента, що містить початковий рядок:

Приклад 11.63.

```
import re
p = re.compile(r"[0-9]+")
print(p.split("word, word\nword"))
```

Результат виконання: `['word, word\nword']`

Функція **`re.split()`**

Формат функції:

re.split(<Шаблон>, <Початковий рядок>[, <Ліміт>[, flags=O]])

Приклад 11.64

```
import re
p = re.compile(r"[\s,.]+")
print(re.split(p, "word1, word2\nword3"))
print(re.split(r"[\s,.]+", "word1, word2\nword3"))
```

Результат виконання:

```
['word1', 'word2', 'word3']
['word1', 'word2', 'word3']
```

Функція **`re.escape()`**

Функція `escape(<Рядок>)` дозволяє екранувати всі спеціальні символи в рядку, отриманому від користувача. Цей рядок надалі можна безпечно використовувати всередині регулярного виразу.

Приклад 11.65.

```
import re
print(re.escape(r"[]().*"))
```

Результат виконання:

```
\[\]\(\)\.\*
```

Функція **`re.purge()`**

Функція `purge()` виконує очищення кеша, у якому зберігаються проміжні дані, використовувані в процесі виконання регулярних виразів. Її рекомендують викликати після обробки великої кількості регулярних виразів. Ця функція не повертає ніякого результату.

Приклад 11.66.

```
import re
re.purge()
```

Контрольні запитання до розділу 11.

1. Дайте визначення регулярних виразів.
2. Який модуль необхідно завантажити для використання регулярних виразів?
3. Які основні методи використовують для створення регулярних виразів?
4. Опишіть використання функції re. findall().
5. Вкажіть основні метасимволи регулярних виразів та поясніть їх дію.
6. Які існують спеціальні послідовності з метасимволом \ ?
7. Опишіть найважливіші прапори компіляції регулярних виразів.
8. Які основні методи об’єкта пошуку вам відомі і які способи їх застосування?
9. Опишіть основні підходи до зворотного посилання на фрагмент.
10. Опишіть способи застосування методу re.split()/