

# ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ Python

# Урок 7

## Файли

### ЗМІСТ

<b>1. Файли .....</b>	<b>3</b>
1.1. Файлова система .....	3
1.2. Типи файлів, текстові та бінарні.....	12
1.3. Робота з файлами .....	15
<b>2. Менеджер контексту .....</b>	<b>31</b>
<b>3. Практичні приклади використання .....</b>	<b>36</b>
3.1. Приклад 1: пошук та заміна слів у текстовому файлі .....	36
3.2. Приклад 2: підрахунок кількості слів у текстовому файлі, які не є числами.....	38
3.3. Приклад 3: вивести слова вмісту файлу у зворотному порядку .....	40
3.4. Приклад 4: видалення заданого по номеру рядка з файлу .....	43

# 1. Файли

## 1.1. Файлова система

*Файл* — це названа область (сукупність даних), розташована у зовнішній пам'яті (на зовнішньому пристрої), і володіє наступними параметрами:

- ім'я файлу (на певному диску), яке дозволяє програмам ідентифікувати файл та, за необхідності, працювати одночасно з кількома файлами;
- довжина файлу може бути обмежена лише обсягом диска.

Ім'я файлу складається з його назви і розширення (наприклад, *myData.txt* і *myData.dat* — різні файли). Чутливість до регістру під час роботи з іменами файлів залежить від операційної системи, що використовується, наприклад, Windows верхній і нижній регістр в іменах файлів і папок сприймає як одне й те саме, а от в Linux використовується файлова система, яка чутлива до регістру.

Дуже поширеною операцією є отримання необхідних даних з файлу або збереження (запис) результатів, отриманих під час роботи програми, у файл. Повне ім'я файлу являє собою повний шлях до каталогу з файлом, що включає ім'я файлу.

Працюючи з файлами, наші дані зберігаються не в оперативній пам'яті (наприклад, як із використанням масивів), а поза нею (наприклад, на жорсткому диску або флеш-накопичувачі), що забезпечує довготривале

зберігання даних за межами програми. У разі вимкнення живлення, пошкодження даних у файлі не відбудеться. Навіть якщо у той момент програма не виконувала над ними операції.

Розмір файлу не є фіксованим, тобто може зменшуватися та збільшуватися. Файли забезпечують зберігання даних довільного типу (тексту, графіки, відео та звуку, виконуваних програм тощо). Для операційної системи файл є такою ж важливою сутністю, як і для її користувача, оскільки усі дані обов'язково мають знаходитися всередині якогось файлу. Інакше операційна система, і, як наслідок, усі її користувачі, не матимуть доступу до цих даних. Робота з усіма пристроями здійснюється також операційною системою за допомогою спеціальних файлів.

Звичайно, файли, за допомогою яких відбувається взаємодія операційної системи та пристроїв комп'ютера, відрізняються від файлів з даними користувача. У спеціальних файлах пристроїв містяться дані, які необхідні операційній системі для взаємодії з такими пристроями, як диски, принтери і т. д.

Незалежно від того, яка операційна система використовується на комп'ютері, файли бувають текстові та бінарні (двійкові). Особливості цих категорій файлів та принципи роботи з ними ми розглянемо трохи згодом.

Також файли поділяються на *виконувані* (програми, які можуть запускатися користувачами, операційною системою або іншими програмами) та *невиконувані* (файли, в яких зберігаються дані). Вміст невиконуваних файлів даних, може змінюватися в процесі роботи (виконання) програм (виконуваних файлів).

Усі файли можна класифікувати таким чином:

- **основні** — наявність яких є обов'язковою для функціонування операційної системи або іншого програмного забезпечення;
- **службові** — відповідальні за різні конфігурації основних файлів;
- **робочі** — з вмістом яких працюють основні файли, змінюючи та створюючи дані;
- **тимчасові** — іноді створюються в процесі роботи деяких основних файлів для зберігання проміжних результатів їх роботи.

На будь-якому носії інформації (жорсткому диску або флеш-накопичувачі) завжди зберігається досить велика кількість файлів. На кожному носії інформації (гнучкому, жорсткому або лазерному диску) може зберігатися багато файлів. Порядок зберігання та спосіб організації файлів на диску визначається файловою системою.

Файлова система є частиною операційної системи, яка надає інтерфейс для взаємодії з різними даними (файлами), що зберігаються на дисках. Можна сказати, що до файлової системи деякого носія інформації (диска) входять:

- **усі файли** на цьому диску;
- **структури даних**, що визначають спосіб організації файлів на диску і порядок роботи з ними (наприклад, каталоги);
- **спеціальні таблиці**, що зберігають інформацію про розподіл вільного та зайнятого простору на диску;
- **системне програмне забезпечення** для керування та роботи з файлами (створення, видалення файлів, читання та запис, пошук тощо).

Фактично саме файлова система пов'язує фізичний носій інформації та програму, яка з нею взаємодіє. Коли деяка програма в процесі своєї роботи звертається до файлу, тоді єдина (базова) інформація, яка їй потрібна — це шлях до файлу та ім'я файлу (іноді його атрибути, дата створення або розмір). Цю інформацію забезпечує драйвер файлової системи.

За аналогічним принципом відбувається і запис даних: програма передає файловій системі базову інформацію про файл, а процес запису та збереження забезпечується файловою системою, відповідно до її правил (алгоритмів роботи).

Таким чином, файлова система забезпечує виконання наступних завдань:

- розподіл файлів на фізичному носії інформації та їх каталогізація;
- реалізація процесів створення, видалення, читання, запису, пошуку файлів;
- зміна атрибутів файлів: назви, розміру, рівня (прав) доступу тощо;
- захист файлів та відновлення інформації в них у випадках збоїв роботи операційної системи.

Майже завжди файли, розташовані на якомусь носії інформації, впорядковуються у каталоги (іноді їх називають директоріями). Можна сказати, що диск складається з двох областей: каталог та область, в якій зберігаються файли.

З погляду користувача, *каталог* — це деяка група файлів, об'єднана самим користувачем або операційною системою за якимись принципами (наприклад, каталог «Мої документи» призначений для зберігання документів

поточного користувача системи). Проте, власне каталог являє собою файл, в якому міститься системна інформація: список файлів, що входять до складу цього каталогу, інформація про розміщення файлів (початок кожного файлу на диску), дані про відповідність файлів їх характеристикам.

Інформація, що зберігається на будь-якому носії, розміщується в кластерах — спеціальних осередках. Якщо розмір файлу не перевищує розмір кластера, то файл буде розміщений файловою системою в одному кластері. Інакше — файл займатиме декілька кластерів.

Уявімо, що наш накопичувач (диск) — це книга зі змістом і пронумерованими сторінками. Тоді весь вміст книги (крім змісту) — це область зберігання файлів, зміст книги — каталог, а сторінки книги — це кластери диска.

Як у змісті книги, навпроти кожної глави розміщений номер сторінки, де й починається ця глава. Так у каталозі, навпроти імені кожного файлу, зберігається номер сектора, з якого починається його розміщення на диску.

Зазвичай на дисках зберігається велика кількість файлів: сотні або навіть тисячі. І використовувати лише один рівень каталогу дуже неефективно.

На рисунку 1 наведено приклад однорівневого каталогу.

C:  
Wiki.txt  
Tornado.jpg  
Notepad.exe

*Рисунок 1*

У цьому випадку всі файли перебувають у кореневому каталозі, тобто безпосередньо на диску.

*Кореневим каталогом* називають базовий (початковий) каталог в структурі носія інформації, в якому можуть розташовуватися як файли, так і інші каталоги. Вкладені каталоги, які зберігаються всередині інших, часто називають *підкаталогами*.

Кожен накопичувач інформації має власне логічне ім'я (зазвичай жорсткі диски мають імена *C:*, *D:* і т. д., флеш-накопичувачі *E:*, *F:*, для даного прикладу). Логічне ім'я надається диску в момент його форматування.

Ім'я кореневого каталогу завжди збігається з ім'ям диска. Коли кількість файлів у користувача починає збільшуватися, процес виявлення потрібної інформації стає важким і витратним за часом. У такій ситуації корисно розподілити всі файли за каталогами за тематикою вмісту або за якимись іншими ознаками.

Тому більш поширеним та зручним способом організації даних є багаторівнева файлова система, в якій всередині каталогів містяться не тільки файли, а й інші каталоги.

У нашій аналогії з книгою така ієрархічна файлова система схожа на зміст, що містить назву розділів книги, глав всередині них, які, у свою чергу, складаються з оповідань.

Ієрархічна файлова система має деревоподібну структуру. Корінь (початкова вершина дерева) відповідає кореневому каталогу. В інших вершинах дерева знаходяться каталоги, які містять інші каталоги та файли.

Кореневі каталоги різних дисків можуть утворювати окремі (незалежні) дерева (така організація існує в операційній системі Windows).



```

C:
  \Program files
    \CDEx
      \CDEx.exe
      \CDEx.hlp
      \mpenc.exe
D:
  \Music
    \ABBA
      \1974 Waterloo
      \1976 Arrival
        \Money, Money, Money.ogg
      \1977 The Album

```

Рисунок 2

Або об'єднуватися в одне (загальне для всіх кореневих дисків) дерево (UNIX-подібні операційні системи).

```

/
  /usr
    /bin
      /arch
      /Is
      /raw
    /lib
      /libhistory.so.5.2
      /libgpm.so.1
  /home
    /lost+found /host.sh /guest
    /Pictures
    /example.png
  /Video
    /matrix.avi
    /news
    /lost_ship.mpeg

```

Рисунок 3

**Примітка:** у файлових системах операційної системи Windows та UNIX-подібних операційних системах використовуються різні типи «слешів»: зворотний слеш «\» для Windows і звичайний слеш «/» для UNIX.

Символи «слешів» використовуються як роздільники між назвами каталогів, які становлять шлях до файлу.

Розглянемо з прикладу файлу «CDEx.exe» (див. Рис. 1). В записі `C:\Program files\CDEx\CDEx.exe`: `C:\Program files\CDEx\` — це шлях до файлу «CDEx.exe».

Можна сказати, що кожна програма працює у певному каталозі файлової системи (поточний або робочий каталог), який є її «робочим місцем». І доступ до даних програма отримує завжди зі свого «робочого місця» (часто програми за замовчуванням працюють з файлами зі свого робочого каталогу).

В ієрархічних файлових системах шлях до файлу може бути повним (абсолютним) або відносним.

Повний (абсолютний) шлях до файлу вказує на певне (завжди однакове) місце у файловій системі. При цьому поточне положення користувача (або програми) в системі ні на що не впливає. Початок повного шляху завжди має ім'я кореневого каталогу.

Відносний шлях — це шлях до файлу відносно поточного (робочого) каталогу (програми) користувача.

Відносний шлях формується так само, як і абсолютний, з використанням та переліченням через «/» усіх назв каталогів, яке зустрічаються при проході до потрібного файлу (каталогу).

Припустимо, що ми запускаємо деяку програму з каталогу */guest*, яка звертається до файлу «*example.png*».

```
/home
  /lost+found
    /host.sh
  /guest
    /Pictures
      /example.png
    /Video
      /matrix.avi
      /news
        /lostship.mpeg
```

Рисунок 4

У цьому випадку робочим каталогом нашої програми є каталог */guest* і ми можемо отримати доступ до потрібного нам файлу таким рядком адреси: *guest/Pictures/example.png*, яка і є відносним шляхом до файлу «*example.png*» (відносно розташування користувача або програми).

Тому й у зверненні до файлу можна використати повне (абсолютне) ім'я файлу (що складається з повного шляху до файлу і самого імені файлу) або відносне ім'я.

А тепер розглянемо ситуацію, при якій нам потрібно не спускатися, а підніматися по дереву каталогів. Наприклад, наша програма знаходиться в каталозі */lost+found*, а нам потрібно звернутися до файлу, розташованому на рівень вище, тобто в каталозі */home*.

Звичайно, ми зможемо використати абсолютний шлях до потрібного файлу. Проте для формування відносного шляху також є спосіб.

Для позначення батьківського каталогу (всередині якого ми знаходимося) використовується комбінація «крапка-крапка»: «../», а для позначення поточного «крапка»: «./».

Таким чином, перебуваючи в каталозі */home*, ми можемо звернутися до файлу в каталозі */lost+found*, використовуючи відносне ім'я файлу: «../*fileName*».

Тепер розглянемо аналогічний приклад для операційної системи Windows. Припустимо, що наша програма — це файл «*CDEx.exe*» і їй потрібно «звернутися» до файлу, розташованого в каталозі Program files.

```
C:
  \Program files
    \CDEx
      \CDEx.exe
      \CDEx.hip
      \mprenc.exe
```

Рисунок 5

У цьому випадку ми також використовуватимемо відносне ім'я файлу: «..\..\ *fileName*».

## 1.2. Типи файлів, текстові та бінарні

Як ми вже знаємо, робота з усіма пристроями комп'ютера здійснюється операційною системою з допомогою спеціальних файлів. Звичайно, ці пристрої сильно відрізняються один від одного. Однак файлова система перетворює їх на єдиний абстрактний логічний пристрій, названий потоком.

**Потік** визначається як послідовність байтів і не залежить від конкретного пристрою, з яким виробляється обмін (оперативна пам'ять, файл на диску, клавіатура або

принтер). Обмін з потоком для збільшення швидкості передачі даних проводиться, зазвичай, через спеціальну область оперативної пам'яті — буфер. Буфер накопичує байти, і фактична передача даних виконується після заповнення буфера. При введенні це дає можливість виправити помилки, якщо дані з буфера ще не надіслані до програми.

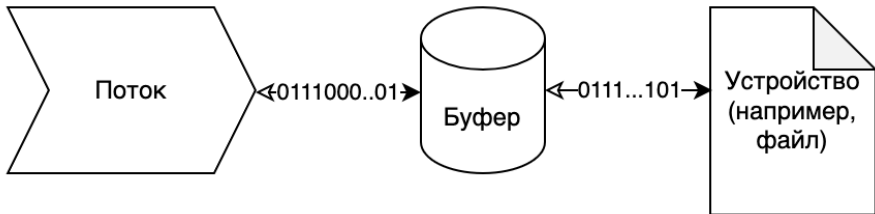


Рисунок 6

**Текстовий потік** — це послідовність символів. Під час передачі символів з потоку на екран, частина з них не виводиться (наприклад, символ повернення каретки, переносу рядка).

**Двійковий потік** — це послідовність байтів, які однозначно відповідають тому, що знаходиться на зовнішньому пристрої.

Припустимо, що наші дані є рядком:

```
s = "Hello\tI likePython\nHi!"
```

При роботі з нею, як із текстовим потоком, послідовності «\t» та «\n» будуть розглядатися як escape-послідовності (з особливостями та призначенням яких ми вже знайомі). Якщо ми обробимо ці дані у бінарному вигляді, то символ «\» буде сприйнятий і оброблений, як звичайний символ.

А рядок «*Hi\xPython*» під час її обробки у текстовому вигляді взагалі стане причиною помилки, оскільки комбінація символів «*\x*» не є escape-послідовністю, а інтерпретатор намагатиметься «зрозуміти» це саме так.

Файли також бувають текстові та двійкові (бінарні). Однак, незалежно від організації даних у файлах, фактично інформація в них представлена в двійковому форматі так, що це ділення умовне. В текстових файлах дані інтерпретуються як послідовність символічних кодів. Спеціальні послідовності використовуються для вказівки ознаки кінця рядка (як у прикладах вище).

Двійкові файли можуть використовуватися для зберігання будь-яких даних (наприклад, зображення у форматі JPEG є двійковим файлом, призначеним для читання програмою роботи з зображеннями). Більшість файлів, які ми бачимо і з якими працюємо за допомогою відповідного програмного забезпечення, є двійковими файлами:

- документи та електронні таблиці: *.pdf*, *.doc*, *.xls* і т. д.
- зображення: *.png*, *.jpg*, *.gif*, *.bmp* і т. д.
- відео: *.mp4*, *.3gp*, *.mkv*, *.avi* і т. д.
- аудіо: *.mp3*, *.wav*, *.mka*, *.aac* і т. д.
- бази даних: *.mdb*, *.accde*, *.frm*, *.sqlite* і т. д.
- архіви: *.zip*, *.rar*, *.iso*, *.7z* і т. д.
- виконувані файли програм: *.exe*, *.dll*, *.class* і т. д.

Дані всередині двійкового файлу зберігаються як необроблені байти, які не можуть бути прочитані людиною.

Звичайно, ми можемо відкрити деякі двійкові файли у звичайному текстовому редакторі, але ми не можемо прочитати вміст, який знаходиться всередині файлу. Це

тому, що усі двійкові файли будуть закодовані у двійковому форматі, який може «зрозуміти» лише комп'ютер (тобто обробити за певними правилами). Якщо відкрити бінарний файл, наприклад, у текстовому редакторі, людина побачить послідовність незрозумілих символів, з яких неможливо отримати суть. Для роботи з такими бінарними файлами нам потрібен певний тип програмного забезпечення, щоб їх відкрити. Наприклад, для відкриття файлу у форматі *.doc* нам потрібен Microsoft Word.

Текстові файли можуть бути відкриті та прочитані людиною у звичайному текстовому редакторі.

Приклади текстових файлів:

- web-документи, стандарти: HTML, XML, CSS, JSON і т. д.
- файли початкових кодів: *c*, *app*, *js*, *py*, *java* і т. д.
- текстові документи: *txt*, *tex*, *RTF* і т. д.
- текстові представлення табличних даних (файли з роздільниками): *csv*, *tsv* і т. д.
- файли налаштувань та конфігурацій: *ini*, *cfg*, *reg* і т. д.

### 1.3. Робота з файлами

Виділяють чотири основні операції під час роботи з файлами:

- Відкриття файлу;
- Читання з файлу;
- Запис у файл;
- Закриття файлу.

У Python для виконання цих операцій є вбудовані функції, що знаходяться в модулі *io*.

За замовчуванням для роботи з файлами використовується модуль *io*, тобто немає потреби імпортувати його перед використанням його функцій. Загальна схема послідовності роботи з файлами представлена на рисунку 7.

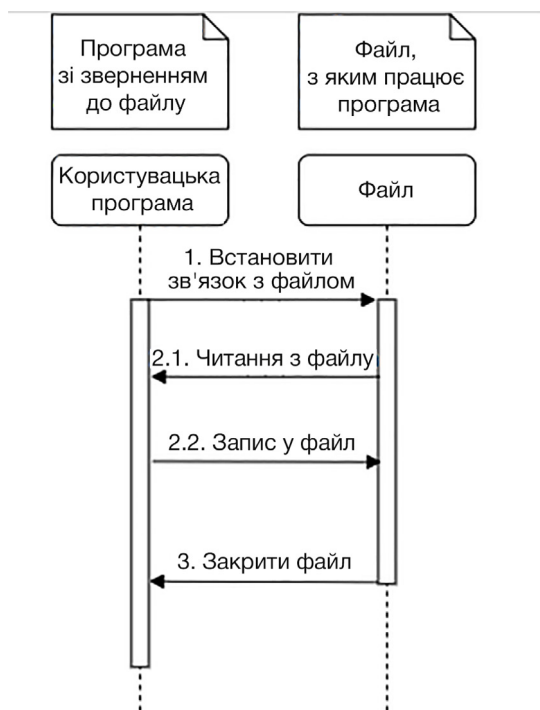


Рисунок 7

Перш за все файл необхідно відкрити, встановивши тим самим зв'язок користувацької програми із зовнішнім носієм даних. Після цього йде власне обробка файлу — запис до нього даних, сформованих у програмі, читання даних з файлу і ряд інших операцій, пов'язаних, наприклад, з пошуком у файлі необхідних



даних. І, нарешті, завершуючи роботу з файлом, його потрібно закрити.

У Python є вбудована функція `open()` для відкриття файлу.

```
fileObj = open(fileName, mode)
```

Дана функція потребує, як мінімум, один аргумент — ім'я файлу. `fileName` — це ім'я файлу (або шлях до нього), який ви хочете відкрити.

**Примітка:** `fileName` має містити ім'я та розширення файлу.

Якщо ми вкажемо лише ім'я файлу (а не шлях до файлу), це означатиме, що програма буде «шукати» такий файл у своєму робочому каталозі.

Другий параметр — `mode` — дозволяє встановити режим відкриття файлу. Розглянемо існуючі режими відкриття файлу (можливі значення параметра `mode`). За замовчуванням значення параметра `mode = 'r'`.

Таблиця 1. Режими відкриття файлів

Режим	Опис	Обробка даних починається з...
<code>r</code>	Відкриття текстового файлу лише для читання. Якщо такого файлу не існує, буде згенеровано виняток	Початку файлу
<code>w</code>	Відкриття текстового файлу лише для запису. Якщо такий файл не існує, він буде створений. Інакше його вміст буде видалено і файл буде перезаписано	Початку файлу
<code>a</code>	Відкриття текстового файлу для додавання. Якщо такий файл не існує, він буде створений	Кінця файлу

Ре- жим	Опис	Обробка даних починається з...
r+	Відкриття текстового файлу для читання та запису. Якщо такого файлу не існує, буде виведена помилка	Початку файлу
w+	Відкриття текстового файлу для читання та запису. Якщо такий файл не існує, він буде створений. Інакше його буде видалено і файл буде перезаписано	Початку файлу
a+	Відкриття текстового файлу для читання та додавання. Якщо такий файл не існує, він буде створений	Кінця файлу
rb	Відкриття двійкового файлу для читання. Якщо такого файлу не існує, буде виведена помилка	Початку файлу
wb	Відкриття двійкового файлу для запису. Якщо такий файл не існує, він буде створений. Інакше його буде видалено і файл буде перезаписано	Початку файлу
ab	Відкриття двійкового файлу для додавання. Якщо такий файл не існує, він буде створений. Інакше дані з нього будуть видалені	Кінця файлу
wb+	Відкриття двійкового файлу для читання та запису. Якщо такий файл не існує, він буде створений. Інакше його буде видалено і файл буде перезаписано	Початку файлу
ab+	Відкриття двійкового файлу для читання та додавання. Якщо такий файл не існує, він буде створений. Інакше його вміст буде видалено	Кінця файлу

В результаті успішного відкриття файлу ми отримаємо файловий об'єкт (який називають дескриптором, «**handler**», деяка абстракція, використовувана операційною системою), який буде використовуватися далі функціями запису у файл, читання з файлу тощо.

Розглянемо на прикладі: відкриємо файл *test.txt* з каталогу «*Data*», який знаходиться у тому ж батьківському каталозі «*Work*», як і каталог «*Python*» — робочий каталог нашої програми (*example.py*).

```
...
Work
  Data
    test.txt
  Python
    example.py
```

Рисунок 8

У цьому випадку файл знаходиться не в поточному каталозі нашої програми, тому передаємо відносний шлях до файлу як перший аргумент (імені файлу) .

```
fileHandler = open("../Data/test.txt")
if fileHandler:
    print("File is open")
```

У цьому прикладі відносний шлях «*../Data/test.txt*» вимагає від програми *example.py* наступних дій: від поточного положення (у каталозі *Python*) потрібно піднятися в батьківський каталог *Work*, зайти у каталог *Data* (який знаходиться всередині каталогу *Work*) і відкрити файл *test.txt*.

Оскільки під час виклику функції `open()` ми вказали лише один аргумент (ім'я файлу), файл буде відкритий в режимі «тільки для читання» (буде використано значення за замовчуванням параметра `mode`).

Якщо спроба відкрити файл не вдалася (наприклад, файл не може бути відкритий у вказаному режимі), буде згенеровано виняток *IOError*, інакше (при успішному відкритті файлу) ми отримаємо файловий об'єкт.

У прикладі ми перевірили наявність такого об'єкта (успішність процедури відкриття файлу) і вивели повідомлення про це.

При спробі відкрити файл, якого немає за вказаним місцезнаходженням (наприклад, відкриття файлу *test.txt* у поточному каталозі, де його немає) отримаємо виняток **FileNotFoundError**:

```
fileHandler = open("test.txt")
               # open file in current directory
```

No such file or directory: 'test.txt'

Рисунок 9

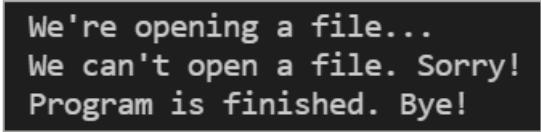
Щоб цей виняток не призводив до раптового (аварійного) завершення програми, нам потрібно його обробити. Для цього є спеціальна конструкція обробки винятків «**try..except**».

```
try:
    print("We're opening a file...")
    fileHandler = open("../Data/test.txt")

except:
    print("We can't open a file. Sorry!")

print("Program is finished. Bye!")
```

У цьому прикладі блок `try` містить фрагмент «небезпечного» коду, який може призвести до генерації виключення (видалимо файл `test.txt` з каталогу `Data`, щоб виконати тестування цієї ситуації).

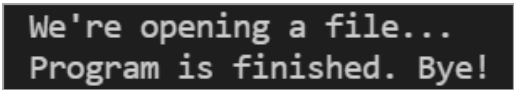


```
We're opening a file...  
We can't open a file. Sorry!  
Program is finished. Bye!
```

Рисунок 10

Блок `except` містить команди, які потрібно виконати у разі виникнення винятку. З таким рішенням, у разі виникнення виняткової ситуації (потрібний файл не було знайдено), наша програма не буде завершуватися аварійно. Для демонстрації цієї особливості ми додали останній рядок коду функцією `print()`, яка виконається у будь-якому випадку.

У разі успішного відкриття файлу, буде виконано лише блок `try` та функція `print()`.



```
We're opening a file...  
Program is finished. Bye!
```

Рисунок 11

Показана в прикладі структура коду дозволяє обробити виняткову ситуацію, якщо вона виникне, без аварійного завершення вже працюючої програми.

Для того, щоб прочитати вміст файлу, нам потрібно відкрити файл у режимі читання. Виконувати читання з файлу у Python можна різними способами, які реалізовані у вигляді методів файлового об'єкта (отриманого в результаті успішного відкриття файлу).

Почнемо з методу `read()`:

```
f.read([size])
```

де

- `f` — ім'я змінної файлового об'єкта;
- `size` — необов'язковий параметр для визначення кількості символів, які будуть прочитані з файлу.

Якщо вам потрібні повністю усі символи з файлу у вигляді рядка, варто використовувати метод `read()` без аргументів.

Давайте прочитаємо весь вміст файлу *test.txt*, який ми відкрили для читання вище:

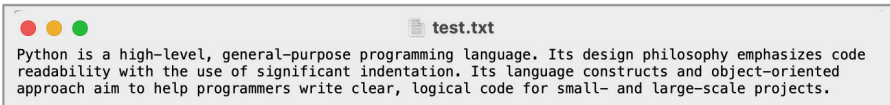


Рисунок 12

```
fileHandler = open("../Data/test.txt")
print (fileHandler.read())
```

```
Python is a high-level, general-purpose programming
language. Its design philosophy emphasizes code
readability with the use of significant indentation.
Its language constructs and object-oriented approach
aim to help prog rammers write clear, logical code
for small- and large-scale projects.
```

Рисунок 13

Якщо нам необхідно прочитати певну кількість символів з файлу (наприклад, 6 символів, — це перше слово

«Python» у нашому файлі), потрібно задати це число як аргумент методу `read()`.

```
fileHandler = open("../Data/test.txt")  
print (fileHandler.read(6)) #Python
```

Після прочитання вказаної кількості символів наступною позицією у файлі, з якої почнеться читання, буде сьомий символ.

Таким чином, якщо у наступному рядку коду ми знову викличемо метод, читання почнеться вже не з початку файлу, а з поточної позиції деякого вказівника, що зміщується зліва направо (від початку файла до кінця) на вказане число позицій (символів).

```
fileHandler = open("../Data/test.txt")  
  
print("1st part:")  
print (fileHandler.read(6))  
print("2nd part:")  
print (fileHandler.read())
```

```
1st part:  
Python  
2nd part:  
is a high-level, general-purpose programming language.  
Its design philosophy emphasizes code readability  
with the use of significant indentation. Its language  
constructs and object-oriented approach aim to help  
programmer s write clear, logical code for small-  
and large-scale projects.
```

Рисунок 14

Тепер розглянемо ситуацію, коли наш файл складається із кількох рядків.

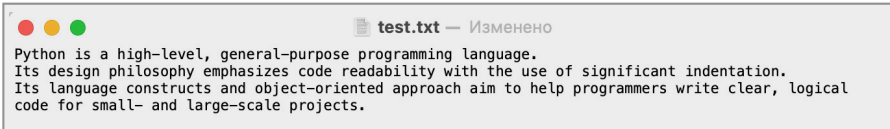


Рисунок 15

Як бачимо на рисунку, при відкритті файлу в текстовому редакторі символи переходу на новий рядок (escape-послідовності «`\n`», з якими ми знайомилися під час роботи з рядками) не відображаються, тобто вони «невидимі» для людини. Однак ми знаємо, що така комбінація символів є в кінці кожного рядка.

Метод `read()` обчислює символи-роздільники рядків як комбінації символів «`\n`» у відповідних позиціях. Для того, щоб побачити цей ефект, скористаємося `raw`-рядками:

```
fileHandler = open("../Data/test.txt")
rawStr=repr(fileHandler.read())
print (rawStr)
```

```
'Python is a high-level, general-purpose programming
language, \nlt design philosophy emphasizes code
readabil ity with the use of significant indentation,
\nlt language constructs and object-oriented
approach
aim to help programmers write clear, logical code for
small- and large-scale projects.'
```

Рисунок 16



А якщо нам потрібно обчислити лише один рядок з файлу? Для цього завдання зручнішим є метод `readline()`. Якщо ми викличемо цей метод без аргументів, то вважаємо один рядок у текстовому файлі, включаючи символи нового рядка «`\n`» (для демонстрації цієї особливості скористаємося raw-рядками, як у попередньому прикладі).

```
fileHandler = open("../Data/test.txt")
str1=fileHandler.readline()
print(str1)
rawStr=repr(str1)
print(rawStr)
```

```
Python is a high-level, general-purpose programming
language.
'Python is a high-level, general-purpose programming
language. \n'
```

Рисунок 17

Після прочитання одного рядка (разом із символом «`\n`») покажчик переходить на початок наступного рядка файлу. Таким чином, повторне використання методу `readline()` у нашому прикладі виконає читання другого рядка файлу:

```
fileHandler = open("../Data/test.txt")

str1=fileHandler.readline()
print(str1)

str2=fileHandler.readline()
print(str2)
```

```
Python is a high-level, general-purpose programming
language.
Its design philosophy emphasizes code readability with
the use of significant indentation.
```

Рисунок 18

Щоб прочитати усі рядки файлу, можна скористатися методом `readlines()`:

```
fileHandler = open("../Data/test.txt")
lines=fileHandler.readlines()
print(lines)
```

```
['Python is a high-level, general-purpose programming
language. \n , Its design philosophy emphasizes code
rea dability with the use of significant indentation.
\n', 'Its language constructs and object-oriented
approach ai m to help programmers write clear,
logical code for small- and large-scale projects.']
```

Рисунок 19

Як ми помітили, результат роботи методу `readlines()` — це список рядків. Такий формат даних можна обробляти, використовуючи широкий набір корисних методів об'єкта «Список», з якими ми вже знайомі.

Ми також можемо прочитати весь файл (рядок за рядком), перебираючи рядки у циклі `for`:

```
fileHandler = open("../Data/test.txt")
for line in fileHandler:
    print(line)
```

Python is a high-level, general-purpose programming language.

Its design philosophy emphasizes code readability with the use of significant indentation.

Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.

Рисунок 20

Для того, щоб виконати запис до файлу, нам потрібно відкрити його в режимі запису «**w**» або додавання «**a**». Будьте уважні використовуючи режим «**w**», тому що в цьому випадку відбудеться перезапис існуючого файлу (тобто втрата усіх попередніх даних).

Запис рядка або послідовності байт виконується за допомогою методу `write()`. Результат роботи методу — кількість символів (або байт), записаних у вказаний файл.

```
fileHandler = open("../Data/newTest.txt","w+")
n=fileHandler.write("How to Create a Text File in Python?")
print("We wrote {} symbols. Let's check it.".format(n))
print(len("How to Create a Text File in Python?"))
```

We wrote 36 symbols. Let's check it  
36

Рисунок 21

У цьому прикладі файлу `newTest.txt` по вказаному шляху не існувало, тому він був створений. В результаті роботи нашого коду зміст цього файлу став таким:

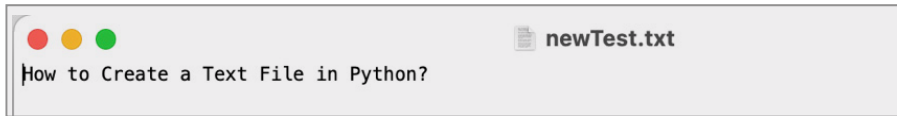


Рисунок 22

Тепер давайте повторно відкриємо цей файл у режимі «W» і за допомогою циклу запишемо в нього п'ять рядків:

```
fileHandler = open("../Data/newTest.txt", "w")
for i in range(5):
    fileHandler.write("This is line %d\n" % (i+1))
```



Рисунок 23

Як бачимо, попередній вміст файлу було повністю перезаписано новим контентом.

Для запису кількох рядків, файл є окремим методом `writelines(listOfStr)`. Цей метод записує вміст списку (який ми передаємо як аргумент) у файл.

Давайте скористаємося ним для додавання (не перезапису) нового контенту (двох рядків, що зберігаються у списку) в наш файл `newTest.txt`. Для цього відкриємо файл у режимі додавання «a».

```
fileHandler = open("../Data/newTest.txt", "a")
myStrs=["Appended line 1\n", "Appended line 2\n"]
fileHandler.writelines(myStrs)
```



Рисунок 24

**Примітка:** якщо ми не вставимо символ «\n» у рядок (у тому місці, де має початися новий), дані будуть послідовно записуватися у текстовий файл, наприклад «Appended line 1Appended line 2».

Після виконання усіх необхідних операцій з файлом, нам потрібно правильно закрити файл. Закриття файлу звільнить ресурси, які були пов'язані з ним. Для вирішення цього завдання використовується метод `close()`:

```
fileHandler = open("../Data/newTest.txt")  
#File Handling  
fileHandler.close()
```

Проте, цей підхід не дуже безпечний. Якщо виникає виняток під час виконання операції з файлом, код завершує роботу, не закриваючи файл. Наприклад, файл був успішно відкритий, але потім (поки що наша програма все ще працює) файл був випадково вилучений. У такій ситуації спроба програми виконати якусь операцію з файлом (наприклад, читання чергової порції даних) викличе генерацію виключення.

Використовуючи «`try-finally`», ми гарантуємо, що файл буде правильно закритий, навіть якщо виникне виняток, який призведе до зупинки виконання програми:

```
try:
    fileHandler = open("../Data/newTest.txt")
    # perform file operations
finally:
    fileHandler.close()
```

Є ще один спосіб роботи з файлом — використання оператора `with`.

```
with open("../Data/newTest.txt") as f:
    # perform file operations
```

Такий підхід гарантує, що файл буде закрито під час виходу з блоку всередині оператора `with`. Нам не потрібно явно викликати метод `close()`. Це робиться всередині блоку неявно.

## 2. Менеджер контексту

У попередньому розділі ми використовували оператор `with` для забезпечення безпечної роботи з файлами навіть у разі виникнення винятків. Отже, фактично перше знайомство з менеджером контексту вже відбулося.

У багатьох завданнях нам необхідно реалізовувати взаємодію із зовнішніми ресурсами: файлами або базами даних. В кожній мові програмування передбачено засоби для реалізації цієї взаємодії. Неконтрольоване використання ресурсів може негативно вплинути на роботу додатка. Наприклад, ми відкрили файл *example.txt*, потім змінили його вміст, додавши новий фрагмент (рядок «*Hello Again!*»), але не закрили його після виконання дій. Припустимо, що потрібно прочитати дані з цього ж файлу. Ми знову відкриваємо його і виконуємо операцію читання. Однак у цьому випадку ми не побачимо внесених змін (нових даних), які ми додали раніше.

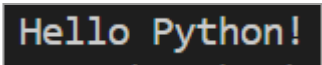
Початковий вміст файлу «*example.txt*».



Рисунок 25

```
fileHandler = open("../Data/example.txt", "a")
fileHandler.write("\nHello Again!")
fileHandler1 = open("../Data/example.txt")
print(fileHandler1.read())
```

Результат операції читання вмісту файлу:



```
Hello Python!
```

Рисунок 26

Як видно з отриманих результатів роботи команди `print()`, ми вивели лише вихідний вміст файлу. А доданий рядок «*Hello Again!*» не було виведено, тобто не був прочитаний програмою. Ця відсутність доступу до оновлених (але ще не збережених) даних сталася через відсутність операції закриття файлу перед його повторним відкриттям.

При роботі з ресурсами (файлом у нашому випадку), однією з основних завдань є звільнення ресурсів після їх використання. Інакше відбудеться або витік інформації (як у наведеному прикладі), або втрата швидкодії, або взагалі збій роботи програми.

Менеджери контексту дозволяють виділяти та звільняти ресурси за потребою. Вони автоматизують етапи встановлення з'єднання з ресурсом і відключення від нього (звільнення ресурсу) щоразу, коли ми у програмі маємо справу із зовнішніми ресурсами. Можна сказати, що менеджери контексту полегшують правильну обробку ресурсів. Даваймо розглянемо використання менеджерів контексту в завданнях обробки файлів.

Раніше ми вже використовували конструкцію «`try-except-finally`» для гарантії закриття файлу навіть у разі виникнення виключення. Однак використання менеджера контексту з ключовим словом `with` є більш простим і компактним способом керування ресурсами. Припустимо, що у нас є дві пов'язані операції, які ми хочемо виконати



в парі (відкриття файлу та його закриття) та блок коду, який має бути виконаний між ними (операції по роботі з файлом). Для реалізації цього механізму використовується така конструкція на основі оператора **with**:

```
with expression as varName:
    Operation1 with varName
    Operation2 with varName
    .....
    OperationN with varName
```

Об'єкт менеджера контексту є результатом виконання виразу **expression** після ключового слова **with**. І далі, в блоці, цей об'єкт доступний за ім'ям **varName**.

Наприклад:

```
with open('../Data/example.txt', 'w') as fileHandler:
    fileHandler.write('Hello!!!')
```

Тут інструкція **open('some\_file', 'w')** — це вираз (**expression**), результатом якого є об'єкт, доступний далі за ім'ям **opened\_file**. Цей фрагмент коду відкриває файл, записує до нього дані і закриває файл після цього. При цьому, навіть у разі виключення (наприклад, помилка запису у файл), менеджер контексту закриє файл.

Всередині блоку **with** можуть знаходитися будь-які (необхідні для реалізації завдання) синтаксичні конструкції. Наприклад, виведемо кожний рядок вмісту файлу:

```
with open('../Data/example.txt', 'r') as fileHandler:
    for line in fileHandler:
        print(line)
```

Наведений вище код відкриє файл і триматиме його відкритим (поки відбувається зчитування та виведення рядків вмісту) до виходу з блоку оператора `with`, після чого файл буде закрито.

Можна використовувати скорочену форму конструкції без `as`-частини. Файлові об'єкти (дескриптори, отримані внаслідок успішного відкриття файлу) є менеджерами контексту. Ми можемо отримати такий об'єкт заздалегідь (до блоку `with`) та продовжити з ним роботу в такий спосіб:

```
fileHandler = open('../Data/example.txt')
with fileHandler:
    fileContents = fileHandler.read()
    print(fileContents)
```

Після завершення операцій з роботи над файлом (які знаходяться всередині блоку `with`) менеджер контексту, як і в попередніх прикладах, закриє файл. У тому числі й у разі виключення.

Python підтримує можливість створення кількох менеджерів контексту в одному операторі `with`. У цьому випадку їх прописують через кому після оператора `with`:

```
with A() as a, B() as b:
    some actions
```

Це станеться в нагоді, коли нам знадобиться відкрити, наприклад, два файли одночасно: перший — для читання, а другий — для запису:

```
path1='../Data/example.txt'
path2='../Data/result.txt'
```

```
with open(path1) as inFile, open(path2, "w") as outFile:  
    # read the content from example.txt  
    fileContents = inFile.read()  
    # transform the content  
    fileContents=fileContents.lower()  
    # write the transformed content to result.txt  
    outFile.write(fileContents)
```

## 3. Практичні приклади використання

Тепер, коли ми вже знайомі з усіма основними операціями по роботі з файлами у Python, давайте розглянемо їх практичне застосування.

### 3.1. Приклад 1: пошук та заміна слів у текстовому файлі

Припустимо, що в нас є текстовий файл `PythonAbout.txt`, що містить такий текст.

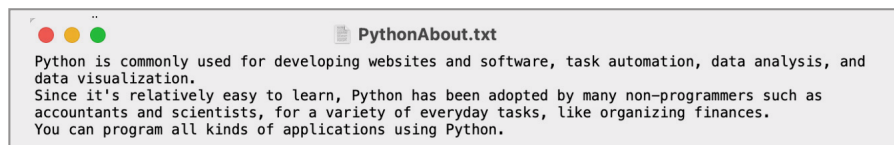


Рисунок 27

І нам необхідно замінити кожне входження слова «*Python*» на «*JavaScript*» у вмісті цього файлу (тричі у нашому прикладі).

Щоб отримати доступ до вмісту файлу, нам потрібно його відкрити. Спочатку ми відкриємо файл у режим читання функцією `open()`. Потім ми виконаємо читання усього вмісту текстового файлу (як одного рядка), використовуючи функцію `read()`. Таким чином, ми зможемо показати (вивести у консоль) вміст файлу до внесення змін. Після виконання цих дій, вміст файлу буде збережено в змінній (тип якого — рядок). Далі застосуємо

до цієї змінної метод рядка `replace()` для заміни одного підрядка («*Python*») на інший («*JavaScript*»).

```
def replaceTextInFile(fileName,originText,newText):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        data = data.replace(originText, newText)

    with open(fileName,'w') as fileHandler:
        fileHandler.write(data)

def readFromFile(fileName):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        print(data)

myFile='../Data/PythonAbout.txt'

print("Original file content:")
readFromFile(myFile)

replaceTextInFile(myFile,'Python','JavaScript')

print("New file content:")
readFromFile(myFile)
```

Результат:

```
Original file content:
Python is commonly used for developing websites and
software, task automation, data analysis, and data
visualiz at ion.
Since it's relatively easy to learn. Python has been
adopted by many non-programmers such as accountants
```

Рисунок 28

```
and scientists, for a variety of everyday tasks,
like organizing finances.
You can program all kinds of applications using
Python.
```

New file content:

```
JavaScript is commonly used for developing websites
and software, task automation, data analysis, and
data visualization.
```

```
Since it's relatively easy to learn, JavaScript
has been adopted by many non-programmers such as
accountants and scientists, for a variety of
everyday tasks, like organizing finances.
You can program all kinds of applications using
JavaScript.
```

Рисунок 28 (продолжение)

### 3.2. Приклад 2: підрахунок кількості слів у текстовому файлі, які не є числами

Досить часто у тексті міститься інформація у вигляді чисел. Припустимо, що нам потрібно підрахувати кількість слів у текстовому файлі, які не є числами.

Вміст нашого файлу «*Info.txt*»:

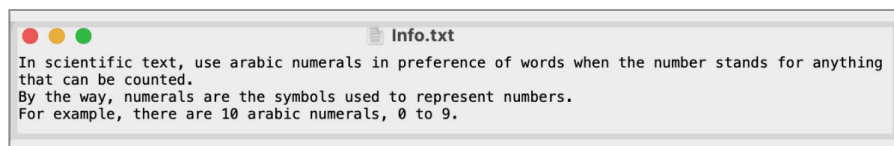


Рисунок 29

Створимо функцію, яка приймає ім'я файлу як аргумент і повертає кількість слів у ньому, які не є числами. В середині функції створимо локальну змінну для зберігання цієї кількості слів і встановимо її початкове значення 0.

Спочатку ми відкриємо наш файл у режимі читання за допомогою функції `open()`. Потім ми виконаємо читання всього вмісту текстового файлу (як одного рядка), використовуючи функцію `read()`. Збережемо вміст в окремій змінній — рядку, який далі розіб'ємо на окремі слова за допомогою функції `split()`. В результаті отримаємо список слів. Далі переберемо усі слова зі списку в циклі, перевіряючи кожне слово за допомогою функції `isnumeric()`. Якщо слово не є числом (результат перевірки функцією `isnumeric()` — `false`), збільшуємо наш лічильник слів на 1.

```
def readFromFile(fileName):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        print(data)

def wordCounter(fileName):
    nWords = 0

    with open(fileName) as fileHandler:
        data = fileHandler.read()
        lines = data.split()
        for word in lines:
            if not word.isnumeric():
                nWords+=1
    return nWords

myFile='../Data/Info.txt'

print("File content:")
readFromFile(myFile)

print("Number of words:{}".format(wordCounter(myFile)))
```

Результат:

```
File content:
In scientific text, use arabic numerals in preference
of words when the number stands for anything that can
be counted.
By the way, numerals are the symbols used to represent
numbers.
For example, there are 18 arabic numerals, e to 9

Number of words: 38
```

Рисунок 30

### 3.3. Приклад 3: вивести слова вмісту файлу у зворотному порядку

Припустимо, що наш текстовий файл *simpleText.txt* містить такий контент:



Рисунок 31

Створимо функцію, яка приймає ім'я файлу як аргумент і виводить слова його вмісту у зворотному порядку.

Спочатку ми відкриємо файл у режим читання, використовуючи функцію `open()`. Потім ми виконаємо читання усього вмісту текстового файлу (як одного рядка), використовуючи функцію `read()`. Збережемо вміст в окремій змінній — рядку, який далі розділимо на окремі слова за допомогою функції `split()`. В результаті отримаємо



список слів. За допомогою вбудованої функції `reversed()`, виконаємо реверс списку слів.

Однак у тексті, крім слів, можуть бути розділові знаки, які в результаті роботи функції `split()` виявляться останнім символом слова (після якого вони стояли у тексті). Тому, перш, ніж виконувати розбиття рядка на слова, необхідно видалити розділові знаки.

Переберемо у циклі всі символи зчитаного з файлу рядка `i`, якщо символ не входить до набору розділових знаків (визначимо цей перелік у вигляді окремого рядка), то поміщаємо його в рядок результату. Реалізуємо для цього завдання окрему функцію.

```
def removePunctuation(myStr, marks):
    resultStr=""
    for symbol in myStr:
        if symbol not in marks:
            resultStr+=symbol
    return resultStr
```

Загальний код програми:

```
def removePunctuation(myStr, marks):
    resultStr=""
    for symbol in myStr:
        if symbol not in marks:
            resultStr+=symbol
    return resultStr
def reverseFileWords(fileName):
    with open(fileName) as fileHandler:
        data = fileHandler.read()
        data=removePunctuation(data, punctuationSymbols)
        words=data.split()
```

```

reversedWords=reversed(words)
for word in reversedWords:
    print(word)

punctuationSymbols='''!()-;?@#$_%:'"\\,./*_'''
myFile='../Data/simpleText.txt'

reverseFileWords(myFile)

```

Результат:

```

visualization
data
and
analysis
data
automation
task
software
and
websites
developing
for
used
commonly
is
Python
Python
Hello

```

Рисунок 32

**Примітка:** якщо немає необхідності видалити розділові знаки з вмісту файлу, потрібно просто видалити виклик функції `removePunctuation()` (рядок коду «`data=removePunctuation(data,punctuationSymbols)`») з тіла функції `reverseFileWords()`.

### 3.4. Приклад 4: видалення заданого по номеру рядка з файлу

Розглянемо на прикладі файлу *Info.txt* такого вмісту:

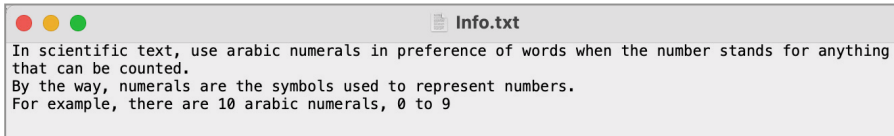


Рисунок 33

І нам необхідно, наприклад, видалити другий рядок у цьому контенті, зберігши отриманий результат у новий файл.

Ми виконаємо читання файлу послідовно рядок за рядком методом `readlines()`. В результаті отримаємо список рядків з вмісту файлу.

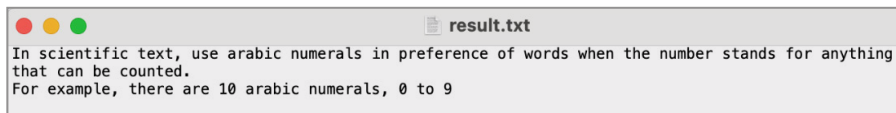
Далі, обробляючи цей список за кожним елементом, будемо перевіряти і порівнювати позицію елемента (рядки файлу) із заданою. Якщо поточний рядок має позицію, яка дорівнює видаленій, вона не записується у текстовий файл з результатами роботи програми.

```
def removeLine(fileIn, fileOut, lineNumber):
    with open(fileIn) as fr:
        lines = fr.readlines()
        counter=1 # position pointer
        with open(fileOut, 'w') as fw:
            for line in lines:
                if counter != lineNumber:
                    fw.write(line)
                counter += 1

myFile='../Data/Info.txt'
```

```
resultFile='../Data/result.txt'  
  
removeLine(myFile, resultFile, 2)
```

Результат:



*Рисунок 34*





## Урок 7

### Файли

© STEP IT Academy, [www.itstep.org](http://www.itstep.org)

© Анна Егошина

Усі права на фото-, аудіо- і відеотвори, що охороняються авторським правом і фрагменти яких використані в матеріалі, належать їх законним власникам. Фрагменти творів використовуються в ілюстративних цілях в обсязі, виправданому поставленим завданням, у рамках учбового процесу і в учбових цілях, відповідно до законодавства про вільне використання твору без згоди його автора (або іншої особи, яка має авторське право на цей твір). Обсяг і спосіб цитованих творів відповідає прийнятим нормам, не завдає збитку нормальному використанню об'єктів авторського права і не обмежує законні інтереси автора і правовласників. Цитовані фрагменти творів на момент використання не можуть бути замінені альтернативними аналогами, що не охороняються авторським правом, і відповідають критеріям добросовісного використання і чесного використання.

Усі права захищені. Повне або часткове копіювання матеріалів заборонене. Узгодження використання творів або їх фрагментів здійснюється з авторами і правовласниками. Погоджене використання матеріалів можливе тільки якщо вказано джерело.

Відповідальність за несанкціоноване копіювання і комерційне використання матеріалів визначається чинним законодавством.