**Модуль 6. Робота з файлами**

Коли застосунок завершує свою роботу, результат його роботи, який зберігався у змінних в застосунку, буде втрачений. Оперативна пам'ять, яка належала застосунку, буде очищена і результат роботи втрачений. Часто нам необхідно зберігати результат роботи, щоб пізніше можна було до нього повернутися. Найпростіший спосіб зберегти корисну інформацію — це записати її у файл на диск.

Python надає функціонал роботи з будь-якими файлами.

**Файлові дескриптори**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D1%96-%D0%B4%D0%B5%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%80%D0%B8)

У Python є абстракція над файлами — це вказівник на файл або файловий дескриптор. Нічого складного в цьому немає, це навпаки спрощує роботу з багатьма системними ресурсами.

Файловий дескриптор — це системний ресурс, доступ до якого надає операційна система. Зазвичай файловий дескриптор можна відкрити (отримати/створити), закрити (повідомити системі, що робота з ним завершена), можна записати у нього щось і прочитати щось.

**Відкриття та закриття файлів**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%B2%D1%96%D0%B4%D0%BA%D1%80%D0%B8%D1%82%D1%82%D1%8F-%D1%82%D0%B0-%D0%B7%D0%B0%D0%BA%D1%80%D0%B8%D1%82%D1%82%D1%8F-%D1%84%D0%B0%D0%B9%D0%BB%D1%96%D0%B2)

Таким чином, робота з файлами у Python починається з відкриття файлу або отримання від системи доступу до файлу, отримання того самого файлового дескриптора. Для цього є вбудована функція **open**, в яку потрібно обов'язково передати ім'я файлу, який ми хочемо відкрити, і можна вказати, як саме ми хочемо відкрити файл:

**fh = open('test\_file.txt')**

В цьому прикладі **fh** — це файловий дескриптор, спеціальний об'єкт, через який ми можемо працювати з файлом.

Після того, як робота з файлом завершена, потрібно повернути ресурс (файл) системі. Для цього у файлового дескриптора потрібно викликати метод **close**:

**fh = open('test.txt')**

**fh.close()**

Закривати файл обов'язково. Незакриті дескриптори можуть стати причиною безлічі неочевидних проблем і складнощів. Найпростіший випадок — це зіпсований файл і повністю втрачена інформація, яка в ньому могла міститися.

Крім того, варто пам'ятати, що оскільки файловий дескриптор — це ресурс, який надається операційною системою, то будь-яке завершення роботи застосунку (аварійне або штатне) не означає закриття усіх відкритих фалів і, знову ж таки, може призвести до помилок.

Якщо не вказати, як ми хочемо відкрити файл, то він відкривається тільки для читання і за допомогою **fh** можна буде тільки читати з файлу. Якщо файлу з ім'ям **test\_file.txt** в системі немає, то ви отримаєте виняток.

Режими відкриття файлів в Python вибираються за допомогою другого аргументу функції **open**.

Можливі режими відкриття файлів:

| **Режим** | **Значення** |
| --- | --- |
| 'r' | відкриття на читання (є значенням за замовчуванням). |
| 'w' | відкриття на запис, вміст файлу видаляється, якщо файлу не існує, створюється новий. |
| 'x' | відкриття на запис, якщо файлу не існує, інакше виняток. |
| 'a' | відкриття на дозапис, інформація додається в кінець файлу. |
| 'b' | відкриття у двійковому режимі. |
| 't' | відкриття в текстовому режимі (є значенням за замовчуванням). |
| '+' | відкриття на читання та запис |

Наприклад, щоб відкрити файл для запису або створити новий, якщо його немає, або перезаписати файл, можна вказати значення **w**:

**fh = open('test.txt', 'w')  
symbols\_written = fh.write('hello!')  
print(symbols\_written) *# 6*  
fh.close()**

В цьому прикладі ми відкрили файл **test.txt** для запису та записали туди рядок **'hello!**' завдовжки 6 символів.

**Читання та запис у файл**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D1%87%D0%B8%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81-%D1%83-%D1%84%D0%B0%D0%B9%D0%BB)

Для запису у файл використовується метод **write** у дескриптора **fh**. Цей метод повертає кількість записаних у файл символів.

Парний до нього метод — це метод read, який дозволяє прочитати деяку кількість символів із файлу.

**fh = open('test.txt', 'w+')  
fh.write('hello!')  
fh.seek(0)  
  
first\_two\_symbols = fh.read(2)  
print(first\_two\_symbols) *# 'he'*  
  
fh.close()**

В цьому прикладі ми відкрили файл для читання та запису. Записали у файл рядок **'hello!'** та прочитали перші два символи із файлу за допомогою методу **read**, вказавши у якості аргументу двійку.

Щоб повернути курсор на початок файлу, викликали метод **seek** та передали йому позицію, куди потрібно переміститися (0).

Щоб прочитати увесь вміст файлу за раз, можна викликати метод **read** без аргументів:

**fh = open('test.txt', 'w')  
fh.write('hello!')  
fh.close()  
  
fh = open('test.txt', 'r')  
all\_file = fh.read()  
print(all\_file) *# 'hello!'*  
  
fh.close()**

Доки файловий дескриптор не закритий, ви можете читати із нього частинами, продовжуючи читання з того самого місця, на якому зупинилися:

**fh = open('test.txt', 'w')  
fh.write('hello!')  
fh.close()  
  
fh = open('test.txt', 'r')  
while True:  
 symbol = fh.read(1)  
 if len(symbol) == 0:  
 break  
 print(symbol)  
  
fh.close()**В цьому прикладі у циклі ми зчитували та виводили у консоль вміст файлу по одному символу за раз. В результаті ви отримаєте у консолі:

**h  
e  
l  
l  
o  
!**

Ще є зручний спосіб читати файл порядково, по одному рядку за раз, для цього можна скористатися методом **readline**:

**fh = open('test.txt', 'w')  
fh.write('first line\nsecond line\nthird line')  
fh.close()  
  
fh = open('test.txt', 'r')  
while True:  
 line = fh.readline()  
 if not line:  
 break  
 print(line)  
  
fh.close()**

В консолі буде виведення:

**first line  
  
second line  
  
third** **line**

Та аналогічний метод **readlines**, який читає увесь файл повністю, але повертає список рядків, де елемент списку — це один рядок

**fh = open('test.txt', 'w')  
fh.write('first line\nsecond line\nthird line')  
fh.close()  
  
fh = open('test.txt', 'r')  
lines = fh.readlines()  
print(lines)  
  
fh.close()**

Виведення у консолі буде:

**['first line\n', 'second line\n', 'third line']**

Зверніть увагу, що всі методи, які читають файли порядково, не опускають (видаляють) символ перенесення рядка.

**Навігація по файлу**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%BD%D0%B0%D0%B2%D1%96%D0%B3%D0%B0%D1%86%D1%96%D1%8F-%D0%BF%D0%BE-%D1%84%D0%B0%D0%B9%D0%BB%D1%83)

Python дає можливість управляти положенням курсора у файлі, можна довільно переміщатися файлом за допомогою методу **seek.** Цей метод приймає один аргумент — це кількість символів, на які потрібно змістити курсор у файлі:

**fh = open('test.txt', 'w+')  
fh.write('hello!')  
  
fh.seek(1)  
second = fh.read(1)  
print(second) *# 'e'*  
  
fh.close()**

В цьому прикладі після запису у файл курсор зупинений на останньому символі. У виразі **fh.seek(1)** ми перемістили курсор на другий символ у файлі.

Переміщаючи курсор, можна перезаписувати символи файлу або читати записане.

Метод **seek** може приймати опціонально другий аргумент, який буде вказувати, звідки потрібно зчитати зміщення. За замовчуванням зміщення задається від початку файлу, але можна задати з кінця або з поточного положення. На жаль, зазначення зміщення з кінця файлу не на всіх платформах підтримується і краще використовувати варіант за замовчуванням.

Щоб дізнатися положення курсора в цей момент, можна скористатися методом **tell**, він повертає позицію (номер) символу з початку файлу, де зараз знаходиться курсор.

**fh = open('test.txt', 'w+')  
fh.write('hello!')  
  
position = fh.tell()  
print(position) *# 6*  
  
fh.seek(1)  
position = fh.tell()  
print(position) *# 1*  
  
fh.read(2)  
position = fh.tell()  
print(position) *# 3*  
  
fh.close()**

**Менеджер контексту**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%BC%D0%B5%D0%BD%D0%B5%D0%B4%D0%B6%D0%B5%D1%80-%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D1%83)

Застосунок може виконати багато операцій між відкриттям та закриттям файлу. В будь-якому місці може статися помилка та застосунок завершиться аварійно, не повернувши файловий дескриптор системі. Така поведінка, як вже згадувалося, небажана і може призводити до втрати даних.

Щоб уникнути цього, можна взяти блок коду, в якому відбувається робота з файлом, у блок **try ... except**:

**fh = open('text.txt')  
try:  
 some\_useful\_function(fh)  
finally:  
 fh.close()**

У цьому прикладі ми викликали функцію **some\_useful\_function** всередині блоку **try ... except** і, якщо станеться виняток, то обов'язково виконається блок **finally**, в якому файл буде закритий. Цей підхід гарантує, що файловий дескриптор буде обов'язково повернений системі.

Але такий підхід не надто елегантний та читабельний.

Для покращення читабельності коду при збереженні функціоналу можна скористатися менеджером контексту **open**. Менеджер контексту — це синтаксична конструкція, яка покращує читабельність коду, але не вносить жодного додаткового функціонала.

**with open('text.txt', 'w+') as fh:  
 some\_useful\_function(fh)**

Менеджер контексту складається з ключового слова **with**, після якого викликається сам менеджер і, якщо щось потрібно повернути з менеджера, то це щось можна передати у змінну, оголошену після ключового слова **as**. Далі ставиться двокрапка і блок коду, який буде виконаний всередині менеджера. У прикладі з **try ... finally** — це код, який йде всередині блоку **try**. Коли код виконається, менеджер контексту виконає те, що повинен зробити в будь-якому випадку, закрити файл, наприклад (це те, що відбувається в блоку **finally**).

Менеджер контексту **open**синтаксично повністю повторює свого класичного тезку **open**, вони повністю ідентичні з точки зору використання.

З точки зору роботи, цей приклад робить у точності те саме, що і попередній з блоком **try ... finally**. Але замість п'яти рядків коду, ви можете написати два, і код виглядає читабельнішим.

Такий спосіб роботи з файлами є рекомендованим у Python, оскільки гарантує, що програміст не забуде закрити файл у будь-якому разі.

**Байт-рядки, масив байтів**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%B1%D0%B0%D0%B9%D1%82-%D1%80%D1%8F%D0%B4%D0%BA%D0%B8-%D0%BC%D0%B0%D1%81%D0%B8%D0%B2-%D0%B1%D0%B0%D0%B9%D1%82%D1%96%D0%B2)

Є ще один контейнер, з яким ми раніше не працювали. Це **bytes** — байтові рядки.

Дані у пам'яті комп'ютера зберігаються у вигляді послідовності байтів. Відповідно, будь-які дані можна представити у вигляді послідовності байтів. Для роботи з "сирими" даними у Python є два типи даних: **bytes** та **bytearray**.

Дуже багато протоколів досі працюють із "сирими" даними або просто потоком байтів, наприклад TCP/IP, послідовний порт, telnet і багато інших.

Щоб працювати з послідовністю байтів у Python є вбудовані типи даних байт-рядків (**bytes**) та масиви байтів **bytearray**.

**bytes**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#bytes)

За своєю суттю байт-рядки або простіше байти — це звичайні рядки, але для запису одного символу використовується суворо один байт.

Байт — це одиниця зберігання та обробки цифрової інформації, що містить 8 біт інформації. Один біт — це 0 або 1. За допомогою одного байта можна записати будь-яке число від 0 до 255 включно.

Для байт-рядків застосовуються ті самі обмеження і правила, що і для звичайних рядків:

* байт-рядки — незмінні;
* байт-рядки — послідовні і до їх елементів можна звертатися за індексом:

**s = b'Hello!'  
print(s[1]) *# b'e'***

* байт-рядки підтримують більшість методів рядків, таких як **upper, startswith, index, find** та інші.

**Створення байт-рядків**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B1%D0%B0%D0%B9%D1%82-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

**byte\_string = b'Hello world!'**В цьому прикладі у **byte\_string** міститься послідовність символів по одному байту на кожного. Від оголошення звичайного рядку байт-рядок відрізняє наявність символу **b** перед самим рядком.

Другий спосіб створення байт рядків — це перетворення у байт-рядок.

Для перетворення рядка у байт-рядок можна скористатися методом рядків encode:

**byte\_str = 'some text'.encode()**У **byte\_str** буде записана послідовність байтів.

В один байт поміщається число від 0 до 255 включно, відповідно можна перетворити послідовність чисел від 0 до 255 у байт-рядок за допомогою функції **bytes**:

**numbers = [0, 128, 255]  
byte\_numbers = bytes(numbers)**

**Виведення байт-рядків**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%B2%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B1%D0%B0%D0%B9%D1%82-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

Для виведення байтів найзручніше скористатися шістнадцятковим записом, в якому для запису чисел від 0 до 255 достатньо двох символів. Саме такий запис використовує Python "за замовчуванням" для байтів.

**some\_numbers = bytes([127, 255, 156])  
print(some\_numbers) *# b'\x7f\xff\x9c'***Символ **\x** вказує на шістнадцятковий формат запису. Щоб перевірити правильність представлення, можна скористатися вбудованою функцією hex, яка перетворить ціле число в рядок — представлення числа в шістнадцятковій формі:

**for num in [127, 255, 156]:  
 print(hex(num))**В результаті виконання ви побачите:

**0x7f  
0xff  
0x9c**

В такому вигляді перші два символи 0x вказують на шістнадцяткову форму запису .

**Кодування рядків (ASCII, UTF-8, CP1251)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%BA%D0%BE%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2-ascii-utf-8-cp1251)

Перші комп'ютери для роботи з текстом використовували так зване ASCII кодування. У цьому кодуванні для запису одного символу використовується один байт.

Зручність цього кодування в тому, що будь-які дані на комп'ютері можна спробувати представити у вигляді тексту в цьому кодуванні. ASCII містить 256 символів. Це не дуже багато і деякий час цього було достатньо. Але з часом алфавіту з 256 символів стало мало, виникла необхідність додати все більше символів (кирилиця, діакритичні знаки, коди валют, ієрогліфи тощо). Щоб задовольнити потребу у додаванні нових символів, придумали використати кодування, де більше одного байту на символ. Python за замовчуванням використовує UTF-8, в якій один символ може займати від 1 до 4 байт, і всього в алфавіті може бути до **1 112 064** знаків. Це не єдине кодування, на різних платформах можуть бути присутні власні, наприклад CP-1251 (кирилиця на ОС сімейства Windows), UTF-16, UTF-32 та інші.

Щоб дізнатися, якому елементу в UTF-8 відповідає символ, є функція **ord** (від order).

Наприклад, символ **'a'** кодується числом 97:

**ord('a') *# 97***

Зворотна операція, коли потрібно дізнатися, який символ закодований числом, наприклад 100, є функція **chr** (скорочено від character):

**chr(128) *# 'd'***

Python може працювати з дуже великою кількістю різних кодувань.

**s = "Привіт!"  
  
utf8 = s.encode()  
print(utf8) *# b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82!'*  
  
utf16 = s.encode('utf-16')  
print(utf16) *# b'\xff\xfe\x1f\x04@\x048\x042\x045\x04B\x04!\x00'*  
  
s\_from\_utf16 = utf16.decode('utf-16')  
print(s\_from\_utf16 == s) *# True***

Спроба перетворити байт-рядок в неправильному кодуванні, призводить або до помилки, або до досить непередбачуваного результату:

**print(b'Hello world!'.decode('utf-16')) *# 效汬⁯潷汲Ⅴ***

**Масив байтів**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%BC%D0%B0%D1%81%D0%B8%D0%B2-%D0%B1%D0%B0%D0%B9%D1%82%D1%96%D0%B2)

Робота з рядками обмежена тим, що рядки і байт-рядки незмінні. Якщо потрібно замінити навіть один символ, потрібно, по суті, створити копію початкового рядка з єдиним відмінним символом. Щоб зменшити накладні витрати при роботі з "сирими" даними, в Python є такий контейнер як **bytearray.**

**byte\_array = bytearray(b'Kill Bill')  
byte\_array[0] = ord('B')  
byte\_array[5] = ord('K')  
print(byte\_array) *# bytearray(b'Bill Kill')***

Основна відмінність від байт-рядків — це змінність, щоб змінити масив байтів, не потрібно створювати новий. Друга важлива відмінність — це те, що масив байтів сприймається системою як послідовність чисел від 0 до 255, а не як послідовність символів в ASCII кодуванні. Саме тому не можна написати **byte\_array[0] = b'B'**. Елементи масиву байтів сприймаються саме як цілі числа.

В іншому ж **bytearray** може використовуватися як заміна байт-рядків і у нього є ті самі методи з тією самою поведінкою.

**Порівняння рядків**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D0%BF%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D1%8F%D0%BD%D0%BD%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

Порівняння рядків у Python може давати неоднозначний результат внаслідок того, що в UTF-8 кодуванні один і той самий символ можна представити декількома кодами, наприклад, символ **'ê'** можна представити кодом U+00EA, або як послідовність двох кодів U+0065 та U+0302. З цієї причини порівняння одного і того самого символу може повернути **False** через відмінності у записі.

Щоб розв'язати цю проблему при роботі з не ASCII символами для порівняння рядків, їх необхідно нормалізувати за допомогою методу **casefold**, який повертає рядок, де всі символи у нижньому регістрі і без неоднозначностей, коли будь-який символ матиме тільки одну можливу форму запису.

**Робота з нетекстовими файлами у Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%BD%D0%B5%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D0%B8%D0%BC%D0%B8-%D1%84%D0%B0%D0%B9%D0%BB%D0%B0%D0%BC%D0%B8-%D1%83-python)

Поки що ми розглядали тільки роботу з текстовими фалами в кодуванні **UTF-8**. Це режим роботи за замовчуванням. Якщо ж потрібно працювати не з текстовими файлами, то можна вказати режим відкриття файлів з **b**, скорочено від **bytes**. У такому режимі ви отримаєте файловий дескриптор для роботи з файлом в режимі байт-рядків.

**with open('raw\_data.bin', 'wb') as fh:  
 fh.write(b'Hello world!')**

В цьому прикладі ми відкрили файл **raw\_data.bin** у режимі для запису "сирих" даних, на що вказує значення wb. В цьому режимі можна писати у файл тільки байт-рядки або байт-масиви.

У режимі роботи з "сирими" даними можна відкрити та прочитати вміст будь-якого файлу, в тому числі й архіву.

**Робота з архівами**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06/#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%B0%D1%80%D1%85%D1%96%D0%B2%D0%B0%D0%BC%D0%B8)

Архіви по своїй суті — це ті самі файли, але інформація в них розташована з використанням алгоритмів стискання, які дозволяють записати інформацію в меншому об'ємі.

Ви можете відкрити будь-який архів як файл в режимі роботи з байт-рядками, реалізувати алгоритм стискання і розпаковування на Python, і отримати власний архіватор/деархіватор. Ця досить цікава вправа і ви, звичайно, можете її виконати, якщо хочете глибше розібратися в алгоритмах стискання.

Одна з головних причин популярності Python — в наявності великої кількості пакетів та модулів зі всіляким функціоналом, які можна використати для своїх потреб. Є простіший спосіб роботи з популярними безкоштовними архівами у Python — це пакет **shutil**, який представляє просунутіший файловий менеджер та вміє працювати з архівами.

**shutil** підтримує архіви **zip, tar, gz**. Для цього він використовує пакети **zipfile** та **tarfile**. Ви можете використовувати їх напряму, якщо хочете.

Щоб запакувати в архів поточну папку, достатньо викликати функцію **make\_archive** пакету **shutil**:

**import shutil  
  
archive\_name = shutil.make\_archive('backup', 'zip')**

Якщо потрібно запакувати іншу папку, можна вказати шлях до папки третім аргументом:

**import shutil  
  
archive\_name = shutil.make\_archive('backup', 'zip', 'some\_folder/inner')**

Обидва виклика створять файл **backup.zip** в поточній робочій папці, а в **archive\_name** буде рядок з повним шляхом до архіву.

Звичайно пакет **shutil** підтримує розпаковування архівів. Для цього є функція **unpack\_archive**, яка розпакує архів у поточну папку або куди вкаже другий аргумент:

**import shutil  
  
archive\_name = shutil.make\_archive('backup', 'zip', 'some\_folder/inner')  
shutil.unpack\_archive(archive\_name, 'new\_folder\_for\_data')**

В цьому прикладі спочатку папка **'some\_folder/inner'** була упакована у **backup.zip**, а потім **backup.zip**був розпакований у папку **'new\_folder\_for\_data'.**

Щоб дізнатися, які формати підтримує пакет і які для них використовуються позначення, можна викликати функцію **get\_archive\_formats.**

**import shutil  
  
for shortcut, description in shutil.get\_archive\_formats():  
 print('{:<10} | {:<10}'.format(shortcut, description))**

У виведенні ви отримаєте таблицю зі скороченням формату архіву та його коротким описом виду:

**"""  
bztar | bzip2'ed tar-file  
gztar | gzip'ed tar-file  
tar | uncompressed tar file  
xztar | xz'ed tar-file  
zip | ZIP file  
"""**

Основна перевага використання **shutil** — це зручний інтерфейс, який візьме на себе рекурсивний прохід по усіх вкладених файлах і папках та збереже структуру файлів та папок, як при архівації, так і при розпаковуванні архіву.