МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра "Захист інформації"

Робота з файлами та обробка винятків

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 8
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Затверджено
на засіданні кафедри
"Захист інформації"
протокол № 01 від 29.08.2024 р.

Львів – 2024

Робота з файлами та обробка винятків: Методичні вказівки до лабораторної роботи № 8 з курсу «Програмування скриптовими мовами» для студентів спеціальності «Кібербезпека» / Укл. *Я. Р. Совин* – Львів: Національний університет "Львівська політехніка", 2024. – 24 с.

Укладач: Я. Р. Совин, канд. техн. наук, доцент

Відповідальний за випуск: В. Б. Дудикевич, д.т.н., професор

Рецензенти: А. Я. Горпенюк, канд. техн. наук, доцент
Ю. Я. Наконечний, канд. техн. наук, доцент

Мета роботи – ознайомитись з роботою з файлами та обробкою виключень у мові Python**.**

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Робота з файлами

Найчастіше дані для обробки надходять із зовнішніх джерел – файлів, у багатьох випадках дані та результати теж потрібно зберегти у файлах. У свою чергу існують різні формати файлів, найбільш простий і універсальний - текстовий. Він відкривається в будь-якому текстовому редакторі. Розширення у текстових файлів: *.txt*, *.html*, *.csv* і т.д. Крім текстових є інші типи файлів – бінарні (*.mp3*, *.mpeg*, *.doc*, *.exe* та ін.), які відкриваються в спеціальних програмах. Розглянемо можливості Python щодо файлових операцій.

1.1.1. Відкриття файлу

Перш ніж працювати з файлом, необхідно створити об'єкт файлу за допомогою функції *open()*. Функція має наступний формат (вказані основні параметри):

```
open(<Шлях до файлу> [, mode = 'r'] [, buffering = -1] [, encoding = None] [, errors = None] [, newline = None])
```

У першому параметрі вказується шлях до файлу. Шлях може бути абсолютним (починаючи з кореневого каталогу) або відносним (відносно поточного робочого каталогу). При вказанні абсолютного шляху в Windows слід враховувати, що в Python слеш є спеціальним символом. З цієї причини слеш необхідно подвоювати або замість звичайних рядків використовувати неформатовані рядки:

```
>>> "C:\\temp\\new\\file.txt" # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt" # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt" # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Зверніть увагу на останній приклад. У цьому шляху через те, що слеш не подвоюється, виникла присутність відразу трьох спеціальних символів: *\t*, *\n* і *\f* (відображається як *\x0c*). Після перетворення цих спеціальних символів шлях буде виглядати наступним чином:

```
C:<Табуляція>emp<Перевід рядка>ew<Перевід формату>ile.txt
```

Якщо такий рядок передати в функцію *open()*, це призведе до генерації винятку *OSError*:

```
>>> open("C:\temp\new\file.txt")
Traceback (most recent call last):
  File "<pyshell#101>", line 1, in <module>
    open("C:\temp\new\file.txt")
OSError: [Errno 22] Invalid argument: 'C:\temp\new\x0cile.txt'
```

Замість абсолютного шляху до файлу можна вказати відносний шлях, який визначається з урахуванням місця розташування діючого робочого каталогу. Відносний шлях буде автоматично перетворений в абсолютний шлях за

допомогою функції *abspath()* з модуля *os.path*.

Можливі наступні варіанти:

♦ якщо файл, що відкривається знаходиться в поточному робочому каталозі, можна вказати тільки ім'я файлу:

```
import os.path # Підключаємо модуль
# Файл в поточному робочому каталозі
# (D:\Python37\Work\Lecture_7\file.txt)
print(os.path.abspath(r"file.txt"))
D:\Python37\Work\Lecture_7\file.txt
```

♦ якщо файл, що відкривається розташований у вкладеному каталозі, перед ім'ям файлу через слеш вказуються імена вкладених каталогів:

```
# Файл, що в D:\Python37\Work\Lecture_7\folder1\
print(os.path.abspath(r"folder1/file.txt"))
D:\Python37\Work\Lecture_7\folder1\file.txt

# Файл, що в D:\Python37\Work\Lecture_7\folder1\folder2\
print(os.path.abspath(r"folder1/folder2/file.txt"))
>>> D:\Python37\Work\Lecture_7\folder1\folder2\file.txt
```

♦ якщо каталог з файлом розташований вище рівнем, перед ім'ям файлу вказуються дві крапки і слеш (*"../"*):

```
# Файл, що в D:\Python37\Work
print(os.path.abspath(r"../file.txt"))
D:\Python37\Work\file.txt
```

♦ якщо на початку шляху розташований слеш, шлях відраховується від кореня диска. В цьому випадку розташування поточного робочого каталогу не має значення:

```
# Файл, що в D:\folder1\file.txt
print(os.path.abspath(r"/folder1/file.txt"))
D:\folder1\file.txt

# Файл, що в D:\folder1\folder2\file.txt
print(os.path.abspath(r"/folder1/folder2/file.txt"))
D:\folder1\folder2\file.txt
```

Як можна бачити, в абсолютному і відносному шляхах можна вказати як прямі, так і зворотні слеші. Всі вони будуть автоматично перетворені з урахуванням значення атрибута *sep* з модуля *os.path*. Значення цього атрибута залежить від операційної системи. Виведемо значення атрибуту *sep* в операційній системі Windows:

```
>>> os.path.sep
'\\'
>>> os.path.abspath(r"D:\Python37/Work\Lecture_7/file.txt")
'D:\\Python37\\Work\\Lecture_7\\file.txt'
```

При використанні відносного шляху необхідно враховувати розташування поточного робочого каталогу, так як робочий каталог не завжди збігається з каталогом, в якому знаходиться виконуваний файл. 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, з якого запускається файл.

Важливо запам'ятати, що поточним робочим каталогом буде каталог, з якого запускається файл, а не каталог, в якому розташований виконуваний файл. Крім

того, шляхи пошуку файлів не мають відношення до шляхів пошуку модулів.

Необов'язковий параметр *mode* у функції *open()* може набувати таких значень:

♦ *r* - тільки читання (значення за замовчуванням). Після відкриття файлу покажчик встановлюється на початок файлу. Якщо файл не існує, генерується виняток *FileNotFoundError*;

♦ *r+* - читання і запис. Після відкриття файлу покажчик встановлюється на початок файлу. Якщо файл не існує, то генерується виняток *FileNotFoundError*;

♦ *w* - запис. Якщо файл не існує, він буде створений. Якщо файл існує, він буде перезаписаний. Після відкриття файлу покажчик встановлюється на початок;

♦ *w+* - читання і запис. Якщо файл не існує, він буде створений. Якщо файл існує, він буде перезаписаний. Після відкриття файлу покажчик встановлюється на початок файлу;

♦ *a* - запис. Якщо файл не існує, він буде створений. Запис здійснюється в кінець файлу. Вміст файлу не видаляється;

♦ *a+* - читання і запис. Якщо файл не існує, він буде створений. запис здійснюється в кінець файлу. Вміст файлу не видаляється;

♦ *x* - створення файлу для запису. Якщо файл вже існує, генерується виняток *FileExistsError*;

♦ *x+* - створення файлу для читання і запису. Якщо файл вже існує, генерується виняток *FileExistsError*.

Після вказівки режиму може слідувати модифікатор:

♦ *b* - файл буде відкритий в бінарному режимі. Файлові методи приймають і повертають об'єкти типу *bytes*;

♦ *t* - файл буде відкритий в текстовому режимі (значення за замовчуванням в Windows). Файлові методи приймають і повертають об'єкти типу *str*. У цьому режимі буде автоматично виконуватися обробка символу кінця рядка - так, в Windows при читанні замість символів *\r\n* буде підставлений символ *\n*. Для прикладу створимо файл *file.txt* і запишемо в нього два рядки:

```
f = open(r"file.txt", "w")  # Відкриваємо файл на запис
f.write("String1\nString2") # Записуємо два рядки в файл
f.close()                   # Закриваємо файл
```

Оскільки ми вказали режим *w*, то, якщо файл не існує, він буде створений, а якщо існує, то буде перезаписаний.

Тепер виведемо вміст файлу в бінарному і текстовому режимах:

```
# Бінарний режим (символ \r залишається)
f = open(r"file.txt", "rb")
for line in f:
    print(repr(line))
f.close()
```

Результат:

```
b'String1\r\n'
b'String2'
# Текстовий режим (символ \r видаляється)
f = open(r"file.txt", "r")
for line in f:
    print(repr(line))
f.close()
```

Результат:

```
'String1\n'
'String2'
```

Для прискорення роботи проводиться буферизація записуваних даних. Інформація з буфера записується в файл повністю тільки в момент закриття файлу або після виклику функції або методу *flush()*. У необов'язковому параметрі *buffering* можна вказати розмір буфера. Якщо в якості значення вказано 0, то дані будуть відразу записуватися в файл (значення допустиме тільки в бінарному режимі). Значення 1 використовується при порядковому запису в файл (значення допустимо тільки в текстовому режимі), інше позитивне число задає приблизний розмір буфера, а негативне значення (або відсутність значення) означає встановлення розміру, заданого в системі за замовчуванням. За замовчуванням текстові файли буферизуются порядково, а бінарні - частинами, розмір яких інтерпретатор вибирає самостійно в діапазоні від 4096 до 8192 байтів.

При використанні текстового режиму (задається за замовчуванням) при читанні проводиться спроба перетворити дані в кодування Unicode, а під час запису виконується зворотна операція - рядок перетвориться в послідовність байтів в певному кодуванні. За замовчуванням призначається кодування, що застосовується в системі. Якщо перетворення неможливо, генерується виняток. Вказати кодування, яка буде використовуватися при записи і читанні файлу, дозволяє параметр *encoding*. Для прикладу запишемо дані в кодуванні UTF-8:

```
f = open (r"file.txt", "w", encoding = "utf-8")
f.write("Рядок")  # Записуємо рядок в файл
f.close()         # Закриваємо файл
```

Для читання цього файлу слід явно вказати кодування при відкритті файлу:

```
f = open(r"file.txt", "r", encoding = "utf-8")
for line in f:
    print(line)
f.close()
# Виведе: Рядок
```

При роботі з файлами в кодуваннях UTF-8, UTF-16 і UTF-32 слід враховувати, що на початку файлу можуть бути присутніми службові символи, 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 в файл під час запису. Щоб символи BOM були додані, в параметрі encoding слід вказати значення *utf-8-sig*. Запишемо рядок в файл в кодуванні UTF-8 з BOM:

```
f = open (r"file.txt", "w", encoding = "utf-8-sig")
f.write("Рядок")  # Записуємо рядок в файл
f.close()         # Закриваємо файл
```

Тепер прочитаємо файл з різними значеннями в параметрі *encoding*:

```
f = open(r"file.txt", "r", encoding = "utf-8")
for line in f:
    print(repr(line))
f.close()
# Виведе: '\ufeffРядок'

f = open(r"file.txt", "r", encoding = "utf-8-sig")
for line in f:
    print(repr(line))
```

```
f.close()
# Виведе: '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 ви не впевнені, 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 BOM 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:

```
f = open (r"file.txt", "w", encoding = "utf-16")
f.write("Рядок")
f.close()

f = open(r"file.txt", "r", encoding = "utf-16")
for line in f:
    print(repr(line))
f.close()
# Виведе: 'Рядок'
```

При використанні значень *utf-16-le*, *utf-16-be*, *utf-32-le* і *utf-32-be* маркер BOM необхідно самим додати в початок файлу, а при читанні видалити його.

У параметрі *errors* можна вказати рівень обробки помилок. Можливі значення: *"strict"* (при помилці генерується виняток *ValueError* - значення за замовчуванням), *"replace"* (невідомий символ замінюється символом питання або символом з кодом *\ufffd*), *"ignore"* (невідомі символи ігноруються), *"xmlcharrefreplace"* (невідомий символ замінюється послідовністю *&#xxxx*;) і *"backslashreplace"* (невідомий символ замінюється послідовністю *\uxxxx*).

Параметр *newline* задає режим обробки символів кінця рядків. Підтримувані ним значення такі:

♦ *None* (значення за замовчуванням) - виконується стандартна обробка символів кінця рядку. Наприклад, в Windows при читанні символи *\r\n* перетворюються в символ *\n*, а під час запису здійснюється 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. В якості спеціального символу можна вказати лише *\r\n*, *\r* і *\n*.

1.1.2. Методи для роботи з файлами

Після відкриття файлу функція *open()* повертає об'єкт, за допомогою якого проводиться подальша робота з файлом. Тип об'єкту залежить від режиму відкриття файлу і буферизації. Розглянемо основні методи:

♦ *close()* - закриває файл. Так як інтерпретатор автоматично видаляє об'єкт, коли на нього відсутні посилання, в невеликих програмах файл можна не закривати явно. Проте, явне закриття файлу є ознакою хорошого стилю

програмування.

Мова Python підтримує протокол менеджерів контексту. Цей протокол гарантує закриття файлу незалежно від того, згенерувався виняток всередині блоку коду чи ні:

```
with open(r"file1.txt", "w", encoding = "cp1251") as f:
    f.write("Рядок") # Записуємо рядок в файл
# Тут файл вже закритий автоматично
```

♦ *write(<Дані>)* - записує дані в файл. Якщо в якості параметра вказана рядок, файл повинен бути відкритий в текстовому режимі, а якщо вказана послідовність байтів - в бінарному. Пам'ятайте, що не можна записувати рядок в бінарному режимі і послідовність байтів в текстовому режимі. Метод повертає кількість записаних символів або байтів. Ось приклад запису в файл:

```
# Текстовий режим
f = open(r"file.txt", "w", encoding = "cp1251")
cnt = f.write("Рядок1\nРядок2") # Записуємо рядок у файл
print("Записано", cnt, "символів")
f.close() # Закриваємо файл

# Бінарний режим
f = open(r"file.txt", "wb")
cnt = f.write(bytes("Рядок1\nРядок2", "cp1251"))
print("Записано", cnt, "байт")
cnt = f.write(bytearray("\nРядок3", "cp1251"))
print("Записано", cnt, "байт")
f.close() # Закриваємо файл
```

Результат:

```
Записано 13 символів
Записано 13 байт
Записано 7 байт
```

Також можна записувати форматовані рядки у файл:

```
file = open('file4.txt', 'w')
file.write("Hello, World!\n")
file.write("Number of entries: %d\nTotal: %8.2f\n" % (1, 3.7))
file.close()
```

Результат:

```
Hello, World!
Number of entries: 1
Total:             3.70
```

Також можливо записувати у файл форматовані рядки функцією *print*:

```
outfile = open('file5.txt', 'w')
print("Hello, World!", file = outfile)
print("Number of entries: %d\nTotal:          %8.2f" % (1, 3.7), file
= outfile)
outfile.close()
```

♦ *writelines(<Послідовність>)* - записує послідовність в файл. Якщо всі елементи послідовності є рядками, файл повинен бути відкритий в текстовому режимі. Якщо всі елементи є послідовностями байтів, то файл повинен бути відкритий в бінарному режимі. Ось приклад запису елементів списку:

```
# Текстовий режим
f = open(r"file.txt", "w", encoding = "cp1251")
f.writelines(["Рядок1\n", "Рядок2"])
```

```
f.close()
# Бінарний режим
f = open(r"file.txt", "wb")
arr = [bytes("Рядок1\n", "cp1251"), bytes("Рядок2", "cp1251")]
f.writelines(arr)
f.close()
```

♦ *writable()* - повертає *True*, якщо файл підтримує запис, і *False* - в іншому випадку:

```
f = open(r"file.txt", "r") # Відкриваємо файл для читання
print(f.writable())        # Виведе: False
f.close()

f = open(r"file.txt", "w") # Відкриваємо файл для запису
print(f.writable())        # Виведе: True
f.close()
```

♦ *read([<Кількість>])* - зчитує дані з файлу. Якщо файл відкритий в текстовому режимі, повертається рядок, а якщо в бінарному - послідовність байтів. Якщо параметр не вказано, повертається вміст файлу від поточної позиції вказівника до кінця файлу:

```
# Текстовий режим
with open(r"file.txt", "r", encoding = "cp1251") as f:
    print(f.read())

# Бінарний режим
with open(r"file.txt", "rb") as f:
    print(f.read())
```

Результат:

```
Рядок1
Рядок2
b'\xd0\xff\xe4\xee\xea1\n\xd0\xff\xe4\xee\xea2'
```

Якщо в якості параметра вказати число, то за кожен виклик буде повертатися вказана кількість символів або байтів. Коли досягається кінець файлу, метод повертає порожній рядок:

```
# Текстовий режим
with open(r"file.txt", "r", encoding = "cp1251") as f:
    print(f.read(7)) # Прочитуємо 7 символів
    print(f.read(6)) # Прочитуємо 6 символів
    print(f.read(6)) # Досягнуто кінець файлу
```

Результат:

```
Рядок1

Рядок2
```

♦ *readline([<Кількість>])* - зчитує з файлу один рядок при кожному виклику. Якщо файл відкритий в текстовому режимі, повертається рядок, а якщо в бінарному – послідовність байтів. Рядок, що повертається включає символ переводу рядка. Винятком є останній рядок - якщо він не завершується символом переведення рядка, то такий доданий не буде. При досягненні кінця файлу повертається порожній рядок:

```
# Текстовий режим
>>> f = open(r"file.txt", "r", encoding = "cp1251")
>>> f.readline(), f.readline()
```

```
('Рядок1\n', 'Рядок2')
>>> f.readline() # Досягнуто кінець файлу
''
>>> f.close()
# Бінарний режим
>>> f = open(r"file.txt", "rb")
>>> f.readline(), f.readline()
(b'\xd0\xff\xe4\xee\xea1\n', b'\xd0\xff\xe4\xee\xea2')
>>> f.readline() # Досягнуто кінець файлу
b''
>>> f.close()
```

Якщо в необов'язковому параметрі вказано число, зчитування буде виконуватися до тих пір, поки не зустрінеться символ нового рядку (*\n*), символ кінця файлу або з файлу не буде прочитано вказану кількість символів. Іншими словами, якщо кількість символів в рядку менша значення параметра, то буде зчитаний один рядок, а не вказана кількість символів, а якщо кількість символів в рядку більше, то повертається вказана кількість символів:

```
>>> f = open(r"file.txt", "r", encoding = "cp1251")
>>> f.readline(2), f.readline(2)
('Ря', 'до')
>>> f.readline(100) # Повертається один рядок, а не 100 символів
'к1\n'
>>> f.close()
```

♦ *readlines()* - зчитує весь вміст файлу в список. Кожен елемент списку буде містити один рядок, включаючи символ переводу рядка. Винятком є останній рядок - якщо він не завершується символом переводу рядка, він доданий не буде. Якщо файл відкритий в текстовому режимі, повертається список рядків, а якщо в бінарному - список об'єктів типу *bytes*:

```
# Текстовий режим
>>> with open(r"file.txt", "r", encoding = "cp1251") as f:
        f.readlines()
['Рядок1\n', 'Рядок2']
# Бінарний режим
>>> with open(r"file.txt", "rb") as f:
        f.readlines()
[b'\xd0\xff\xe4\xee\xea1\n', b'\xd0\xff\xe4\xee\xea2']
```

♦ *__next__()* - зчитує один рядок при кожному виклику. Якщо файл відкритий в текстовому режимі, повертається рядок, а якщо в бінарному - послідовність байтів. При досягненні кінця файлу генерується виняток *StopIteration*:

```
>>> f = open(r"file.txt", "r", encoding = "cp1251")
>>> f.__next__(), f.__next__()
('Рядок1\n', 'Рядок2')
>>> f.__next__() # Досягнуто кінець файлу
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    f.__next__() # Досягнуто кінець файлу
StopIteration
>>> f.close()
```

Завдяки методу *__next__()* ми можемо перебирати файл порядково в циклі *for*. Цикл *for* на кожній ітерації буде автоматично викликати метод *__next__()*. Для

прикладу виведемо всі рядки, попередньо видаливши символ переводу рядка:

```
>>> f = open(r"file.txt", "r", encoding = "cp1251")
>>> for line in f:
        print(line.rstrip("\n"), end = " ")
Рядок1 Рядок2
>>> f.close()
```

♦ *flush()* - примусово записує дані з буфера на диск;

♦ *fileno()* - повертає цілочисельний дескриптор файлу. Значення, що повертається завжди буде більше числа 2, оскільки число 0 закріплено за стандартним вводом *stdin*, 1 – за стандартним виводом *stdout*, а 2 - за стандартним виводом повідомлень про помилки *stderr*:

```
>>> f = open(r"file.txt", "r", encoding = "cp1251")
>>> f.fileno()
3
>>> f.close()
```

♦ *truncate([<Кількість>])* - обрізає файл до зазначеної кількості символів (якщо заданий текстовий режим) або байтів (в разі бінарного режиму). Метод повертає новий розмір файлу:

```
>>> f = open(r"file.txt", "r+", encoding = "cp1251")
>>> f.read()
'Рядок1\nРядок2'
>>> f.truncate(5)
5
>>> f.close()
>>> with open (r"file.txt", "r", encoding = "cp1251") as f:
        f.read()
'Рядок'
```

♦ *tell()* - повертає позицію покажчика щодо початку файлу в вигляді цілого числа. Зверніть увагу: в Windows метод *tell()* вважає символ *\r* як додатковий байт, хоча цей символ видаляється при відкритті файлу в текстовому режимі:

```
>>> with open (r"file.txt", "w",  encoding = "cp1251") as f:
        f.write("String1\nString2")
15
>>> f = open(r"file.txt", "r", encoding = "cp1251")
>>> f.tell()     # Покажчик розташований на початку файлу
0
>>> f.readline() # Переміщаємо покажчик
'String1\n'
>>> f.tell()     # Повертає 9 (8 + '\r'), а не 8 !!!
9
>>> f.close()
```

Щоб уникнути цієї невідповідності, слід відкривати файл в бінарному режимі, а не в текстовому:

```
>>> f = open(r"file.txt", "rb")
>>> f.readline() # Переміщаємо покажчик
b'String1\r\n'
>>> f.tell()     # Тепер значення відповідає
9
>>> f.close()
```

♦ *seek(<Зміщення> [, <Позиція>])* - встановлює покажчик в позицію, що має задане *<Зміщення>* щодо параметра *<Позиція>*. Як параметр *<Позиція>* можуть

бути вказані такі 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. Позитивне значення зсуву викликає перехід у кінець файлу, негативне - до його початку;

• *io.SEEK_END* або 2 - кінець файлу.

Виведемо значення цих атрибутів:

```
>>> import io
>>> io.SEEK_SET, io.SEEK_CUR, io.SEEK_END
(0, 1, 2)
```

Ось приклад використання методу *seek()*:

```
>>> import io
>>> f = open(r"file.txt", "rb")
>>> f.seek(9, io.SEEK_CUR) # 9 байтів від покажчика
9
>>> f.tell()
9
>>> f.seek(0, io.SEEK_SET) # Переміщаємо покажчик в початок
0
>>> f.tell()
0
>>> f.seek(-9, io.SEEK_END) # -9 байтів від кінця файлу
4
>>> f.tell()
4
>>> f.close()
```

♦ *seekable()* - повертає *True*, якщо покажчик файлу можна зрушити в іншу позицію, і *False* - в іншому випадку:

```
>>> f = open(r"file.txt", "rb")
>>> f.seekable()
True
```

Крім методів, об'єкти файлів підтримують кілька атрибутів:

♦ *name* - ім'я файлу;

♦ *mode* - режим, в якому був відкритий файл;

♦ *closed* - повертає *True*, якщо файл був закритий, і *False* - в іншому випадку:

```
>>> f = open(r"file.txt", "r+b")
>>> f.name, f.mode, f.closed
('file.txt', 'rb+', False)
>>> f.close()
>>> f.closed
True
```

♦ *encoding* - назва кодування, яка буде використовуватися для перетворення рядків перед записом у файл або при читанні. Атрибут доступний тільки в текстовому режимі:

```
>>> f = open(r"file.txt", "a", encoding = "cp1251")
>>> f.encoding
'cp1251'
>>> f.close()
```

Стандартний вивід *stdout* також є файловим об'єктом. Атрибут *encoding* цього об'єкта завжди містить кодування пристрою виведення, тому рядок перетвориться в послідовність байтів в правильному кодуванні. Наприклад, при

запуску за допомогою подвійного клацання на значку файлу атрибут *encoding* матиме значення *"cp866"*, а при запуску у вікні Python Shell редактора IDLE - значення *"cp125l"*:

```
>>> import sys
>>> sys.stdout.encoding
'cp1251'
```

♦ *buffer* - дозволяє отримати доступ до буферу. Атрибут доступний тільки в текстовому режимі. За допомогою цього об'єкта можна записати послідовність байтів в текстовий потік:

```
>>> f = open(r"file.txt", "w", encoding = "cp1251")
>>> f.buffer.write(bytes("Рядок", "cp1251"))
5
>>> f.close()
```

1.1.3. Функції для маніпулювання файлами

Для копіювання і переміщення файлів призначені наступні функції з модуля *shutil*:

♦ *copyfile(<Копійований файл>, <Куди копіюємо>)* - дозволяє скопіювати вміст файлу в інший файл. Ніякі метадані (наприклад, права доступу) не копіюються. Якщо файл існує, він буде перезаписаний. Якщо файл не вдалося скопіювати, генерується виняток *OSError* або один з винятків, що є підкласом цього класу. Як результат повертається шлях файлу, куди були скопійовані дані:

```
import shutil # Підключаємо модуль
shutil.copyfile(r"file.txt ", r"file2.txt")
# Шлях не існує:
shutil.copyfile(r"file.txt", r"D:\book2\file2.txt")
Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v2.py", line 227, in <module>
    shutil.copyfile(r"file.txt", r"D:\book2\file2.txt")
  File "D:\Python37\lib\shutil.py", line 121, in copyfile
    with open(dst, 'wb') as fdst:
FileNotFoundError: [Errno 2] No such file or directory:
'D:\\book2\\file2.txt'
```

Виняток *FileNotFoundError* є підкласом класу *OSError* і генерується, якщо зазначений файл не знайдений.

♦ *copy(<Копійований файл>, <Куди копіюємо>)* - дозволяє скопіювати файл разом з правами доступу. Якщо файл існує, він буде перезаписаний. Якщо файл не вдалося скопіювати, генерується виняток *OSError* або один з винятків, що є підкласом цього класу. Як результат повертає шлях скопійованого файлу:

```
>>> shutil.copy(r"file.txt", r"file3.txt")
'file3.txt'
```

♦ *move(<Шлях до файлу>, <Куди переміщаємо>)* - переміщує файл в зазначене місце з видаленням вихідного файлу. Якщо файл існує, він буде перезаписаний. Якщо файл не вдалося перемістити, генерується виняток *OSError* або один з винятків, що є підкласом цього класу. Як результат повертає шлях переміщеного файлу. Ось приклад переміщення файлу *file3.txt* в каталог *D:\book\folder1*:

```
>>> shutil.move(r"file3.txt", r"D:\book\folder1")
```

```
'D:\\book\\folder1\\file3.txt'
```

Для перейменування і видалення файлів призначені наступні функції з модуля *os*:

♦ *rename (<Старе ім'я>, <Нове ім'я>)* - перейменовує файл. Якщо файл не вдалося перейменувати, генерується виняток *OSError* або один з винятків, що є підкласом цього класу. Ось приклад перейменування файлу з обробкою винятків:

```
import os # Підключаємо модуль
try:
    os.rename(r"file2.txt", "file4.txt")
except OSError:
    print("Файл не вдалося перейменувати")
else:
    print( "Файл успішно перейменований")
```

♦ *remove(<Шлях до файлу>)* і *unlink(<Шлях до файлу>)* - дозволяють видалити файл. Якщо файл не вдалося видалити, генерується виняток *OSError* або один з винятків, що є підкласом цього класу.

```
>>> os.remove(r"file3.txt")
>>> os.unlink(r"file4.txt")
```

Модуль *os.path* містить додаткові функції, що дозволяють перевірити наявність файлу, отримати розмір файлу та ін. Наведемо ці функції:

♦ *exists(<Шлях або дескриптор>)* - перевіряє зазначений шлях на існування. Повертає *True*, якщо шлях існує, і *False* - в іншому випадку:

```
>>> import os.path
>>> os.path.exists(r"file.txt"), os.path.exists(r"file55.txt")
(True, False)
>>> os.path.exists(r"D:\book"), os.path.exists(r"D:\book55")
(True, False)
```

♦ *getsize (<Шлях до файлу>)* - повертає розмір файлу в байтах. Якщо файл не існує, генерується виняток *OSError*:

```
>>> os.path.getsize(r"file.txt")
13
```

♦ *getatime(<Шлях до файлу>)* - повертає час останнього доступу до файлу в вигляді кількості секунд, що пройшли з початку епохи (1 січня 1970 року). Якщо файл не існує, генерується виняток *OSError*.

♦ *getctime(<Шлях до файлу>)* - повертає дату створення файлу у вигляді кількості секунд, що минули з початку епохи. Якщо файл не існує, генерується виняток *OSError*.

♦ *getmtime(<Шлях до файлу>)* - повертає час останньої зміни файлу у вигляді кількості секунд, що пройшли з початку епохи. Якщо файл не існує, генерується виняток *OSError*.

```
>>> import time # Підключаємо модуль time
>>> t = os.path.getatime(r"file.txt")
>>> t
1565779485.3148918
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'14.08.2019 13:44:45'
>>> t = os.path.getctime(r"file.txt")
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'14.08.2019 13:44:45'
>>> t = os.path.getmtime(r"file.txt")
```

```
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'14.08.2019 17:02:10'
```

1.1.4. Перетворення шляху до файлу або каталогу

Перетворити шлях до файлу або каталогу дозволяють наступні функції з модуля *os.path*:

♦ *abspath(<Відносний шлях>)* - перетворює відносний шлях в абсолютний, враховуючи розташування поточного робочого каталогу:

```
>>> import os.path
>>> os.path.abspath(r"file.txt")
'D:\\book\\file.txt'
>>> os.path.abspath(r"folder1/file.txt")
'D:\\book\\folder1\\file.txt'
>>> os.path.abspath(r"../file.txt")
'D:\\file.txt'
```

Крім того, якщо слеш розташований в кінці рядка, то його необхідно подвоювати навіть при використанні неформатованих рядків:

```
>>> r"C:\temp\new\"
SyntaxError: EOL while scanning string literal
```

Тому в даному випадку краще використовувати звичайні рядки:

```
>>> "C:\\temp\\new\\"     # Правильно
'C:\\temp\\new\\'
>>> r"C:\temp\new\\"[:-1] # Можна і видалити слеш
'C:\\temp\\new\\'
```

♦ *basename(<Шлях>)* - повертає ім'я файлу без шляху до нього:

```
>>> os.path.basename(r"D:\book\folder1\file.txt")
'file.txt'
>>> os.path.basename(r"D:\book\folder")
'folder'
>>> os.path.basename(r"D:\book\folder\\")
''
```

♦ *dirname(<Шлях>)* - повертає шлях до каталогу, де зберігається файл:

```
>>> os.path.dirname(r"D:\book\folder1\file.txt")
'D:\\book\\folder1'
>>> os.path.dirname(r"D:\book\folder")
'D:\\book'
>>> os.path.dirname(r"D:\book\folder\\")
'D:\\book\\folder'
```

♦ *split(<Шлях>)* - повертає кортеж з двох елементів: шляхи до каталогу, де зберігається файл, та імені файлу:

```
>>> os.path.split(r"D:\book\folder1\file.txt")
('D:\\book\\folder1', 'file.txt')
>>> os.path.split(r"D:\book\folder")
('D:\\book', 'folder')
>>> os.path.split(r"D:\book\folder\\")
('D:\\book\\folder', '')
```

♦ *splitdrive(<Шлях>)* - розділяє шлях на ім'я диску і решту шляху. В якості значення повертається кортеж з двох елементів:

```
>>> os.path.splitdrive(r"D:\book\folder1\file.txt")
('D:', '\\book\\folder1\\file.txt')
```

♦ *splitext(<Шлях>)* - повертає кортеж з двох елементів: шляху з ім'ям файлу, але без розширення, і розширення файлу (фрагмент після останньої крапки):

```
>>> os.path.splitext(r"D:\book\folder\file.tar.gz")
('D:\\book\\folder\\file.tar', '.gz')
```

1.1.5. Збереження об'єктів в файл

Зберегти об’єкти в файл і в подальшому відновити об’єкти з файлу дозволяє модуль *pickle*. Модуль *pickle* надає наступні функції:

♦ *dump(<Об'єкт>, <Файл> [, <Протокол>] [, fix_imports = True])* - здійснює серіалізацію об'єкта і записує дані в зазначений файл. У параметрі *<Файл>* вказується файловий об'єкт, відкритий на запис в бінарному режимі. Ось приклад збереження об'єкта в файл:

```
import pickle
f = open(r"file.txt", "wb")
obj = ["Рядок", (2, 3)]
pickle.dump(obj, f)
f.close()
```

♦ *load()* - читає дані з файлу і перетворює їх в об'єкт. Формат функції:

```
load(<Файл> [, fix_imports = True] [, encoding = "ASCII"] [, errors =
"strict"])
```

У параметрі *<Файл>* вказується файловий об'єкт, відкритий на читання в бінарному режимі. Ось приклад відновлення об'єкту з файлу:

```
import pickle
f = open(r"file.txt", "wb")
obj = ["Рядок", (2, 3)]
pickle.dump(obj, f)
f.close()

f = open(r"file.txt", "rb")
obj = pickle.load(f)
print(obj)
f.close()
```

Результат:

```
['Рядок', (2, 3)]
```

В один файл можна зберегти відразу декілька об'єктів, послідовно викликаючи функцію *dump()*:

```
obj1 = ["Рядок", (2, 3)]
obj2 = (1, 2)
f = open(r"file.txt", "wb")
pickle.dump(obj1, f) # Зберігаємо перший об'єкт
pickle.dump(obj2, f) # Зберігаємо другий об'єкт
f.close()
```

Для відновлення об'єктів необхідно кілька разів викликати функцію *load()*:

```
f = open(r"file.txt", "rb")
obj1 = pickle.load(f) # Відновлюємо перший об'єкт
obj2 = pickle.load(f) # Відновлюємо другий об'єкт
print(obj1, obj2)     # Виведе: ['Рядок', (2, 3)] (1, 2)
f.close()
```

1.1.6. Функції для роботи з каталогами

Для роботи з каталогами використовуються наступні функції з модуля *os*:

♦ *getcwd()* - повертає поточний робочий каталог. 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 виконуваним файлом:

```
>>> import os
>>> os.getcwd() # Поточний робочий каталог
'D:\\Python37\\Work\\Lecture_7'
```

♦ *chdir(<Ім'я каталогу>)* - робить зазначений каталог поточним:

```
>>> os.chdir("D:\\book\\folder1\\")
>>> os.getcwd() # Поточний робочий каталог
'D:\\book\\folder1'
```

♦ *mkdir(<Ім'я каталогу> [, <Права доступу>])* - створює новий каталог з правами доступу, зазначеними в другому параметрі. Ось приклад створення нового каталогу в поточному робочому каталозі:

```
>>> os.mkdir("newfolder") # Створення каталогу
```

♦ *rmdir(<Ім'я каталогу>)* - видаляє порожній каталог. Якщо в каталозі є файли або вказаний каталог не існує, генерується виняток підклас класу *OSError*. Видалимо каталог *newfolder*:

```
>>> os.rmdir("newfolder") # Видалення каталогу
```

♦ *listdir(<Шлях>)* - повертає список об'єктів в зазначеному каталозі:

```
>>> os.listdir("D:\\book\\folder1\\")
['module1.py', '__init__.py', '__pycache__']
```

♦ *walk()* - дозволяє обійти дерево каталогів. Формат функції:

```
walk(<Початковий каталог> [, topdown = True] [, onerror = None] [,
followlinks = False])
```

Як значення функція *walk()* повертає об'єкт. На кожній ітерації через цей об'єкт доступний кортеж з трьох елементів: поточного каталогу, списку каталогів і списку файлів, що знаходяться в ньому. Якщо провести зміни в списку каталогів під час виконання, це дозволить змінити порядок обходу вкладених каталогів.

Необов'язковий параметр *topdown* задає порядок пересування між каталогами: якщо в якості значення вказано *True* (значення за замовчуванням), порядок пересування буде таким:

```
>>> for(p, d, f) in os.walk("D:\\book\\folder1\\"): print(p)
D:\book\folder1\
D:\book\folder1\__pycache__
```

Якщо в параметрі *topdown* вказано значення *False*, порядок пересування буде інший:

```
>>> for(p, d, f) in os.walk("D:\\book\\folder1\\", False): print(p)
D:\book\folder1\__pycache__
D:\book\folder1\
```

Видалити дерево каталогів дозволяє також функція *rmtree()* з модуля *shutil*. Функція має такий вигляд:

```
rmtree(<Шлях> [, <Обробка помилок> [,<Обробник помилок>]])
```

Якщо в параметрі *<Обробка помилок>* вказано значення *True*, помилки будуть проігноровані. Якщо вказано значення *False* (значення за замовчуванням), в третьому параметрі можна задати посилання на функцію, яка буде викликатися

при виникненні винятку.

Ось приклад видалення дерева каталогів разом з початковим каталогом:

```
>>> import shutil
>>> shutil.rmtree("D:\\book\\folder1\\")
```

Перевірити, на який тип об'єкта посилається елемент можна за допомогою наступних функцій з модуля *os.path*:

♦ *isdir(<Об'єкт>)* - повертає *True*, якщо об'єкт є каталогом, і *False* - в іншому випадку:

```
>>> import os.path
>>> os.path.isdir(r"D:\book\file.txt")
False
>>> os.path.isdir("D:\\book\\")
True
```

♦ *isfile(<Об'єкт>)* - повертає *True*, якщо об'єкт є файлом, і *False* - в іншому випадку:

```
>>> os.path.isfile(r"D:\book\file.txt")
True
>>> os.path.isfile ("D:\\book\\")
False
```

1.1.7. Винятки для файлових операцій

Функції і методи, які здійснюють файлові операції, при виникненні нештатних ситуацій генерують виняток класу *OSError* або один з винятків, які є його підкласами. Винятків-підкласів класу *OSError* досить багато. Ось ті з них, що зачіпають саме операції з файлами і каталогами:

♦ *BlockingIOError* - не вдалося заблокувати об'єкт (файл або потік вводу/виводу);

♦ *ConnectionError* - помилка з'єднання з мережею. Може виникнути при відкритті файлу по мережі;

♦ *FileExistsError* - файл або каталог з заданим ім'ям вже існують;

♦ *FileNotFoundError* - файл або каталог з заданим ім'ям не виявлені;

♦ *InterruptedError* - файлова операція несподівано перервана;

♦ *IsADirectoryError* - замість шляху до файлу вказано шлях до каталогу;

♦ *NotADirectoryError* - замість шляху до каталогу вказано шлях до файлу;

♦ *TimeoutError* - минув час, відведений системою на виконання операції.

Ось приклад коду, який займається обробкою деякі із зазначених винятків:

```
try
     open("C:\temp\new\file.txt")
except FileNotFoundError:
     print("Файл відсутній")
except IsADirectoryError:
     print("Це не файл, а каталог")
except OSError:
     print("Невстановлена помилка відкриття файлу")
```

1.2. Обробка винятків

Винятки (ексепшени, except) - це сповіщення інтерпретатора, що генеруються в разі виникнення помилки в програмному коді або при настанні

якої-небудь події. Якщо в коді не передбачена обробка винятків, виконання програми переривається, і виводиться повідомлення про помилку. Якщо в програму включений код *обробки винятків* (exeption handler) – виконання програми продовжиться. Реакція на виняток називається *перехопленням винятку*. Python надає набір винятків для багатьох стандартних ситуацій, а користувачі можуть визначати власні винятки для своїх цілей.

У програмі можуть зустрітися три типи помилок:

♦ *синтаксичні* - це помилки в імені оператора або функції, відсутність закриваючих або відкриваючих лапок і т. д. - тобто помилки в синтаксисі мови. Як правило, інтерпретатор попередить про наявність такої помилки, а програма не виконуватиметься зовсім. Приклад синтаксичної помилки:

```
>>> print("Немає закриваючих лапок!)
SyntaxError: EOL while scanning string literal
```

♦ *логічні* - це помилки в логіці програми, які можна виявити тільки за результатами її роботи. Як правило, інтерпретатор не попереджає про наявність таких помилок, і програма буде успішно виконуватися, але результат її виконання виявиться не тим, на який розраховували. Виявити і виправити логічні помилки досить важко;

♦ *помилки часу виконання* - це помилки, які виникають під час роботи програми. Причиною є події, які не передбачені програмістом. Класичним прикладом є ділення на нуль:

```
>>> def test(x, y): return x / y
>>> test(4, 2) # Нормально
2.0
>>> test(4, 0) # Помилка
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    test(4, 0) # Помилка
  File "<pyshell#24>", line 1, in test
    def test(x, y): return x / y
ZeroDivisionError: division by zero
```

У Python винятки генеруються не тільки при виникненні помилки, але і як повідомлення про настання будь-яких подій. Наприклад, метод *index()* генерує виняток *ValueError*, якщо шуканий фрагмент не входить в рядок:

```
>>> "Рядок".index("текст")
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    "Рядок".index("текст")
ValueError: substring not found
```

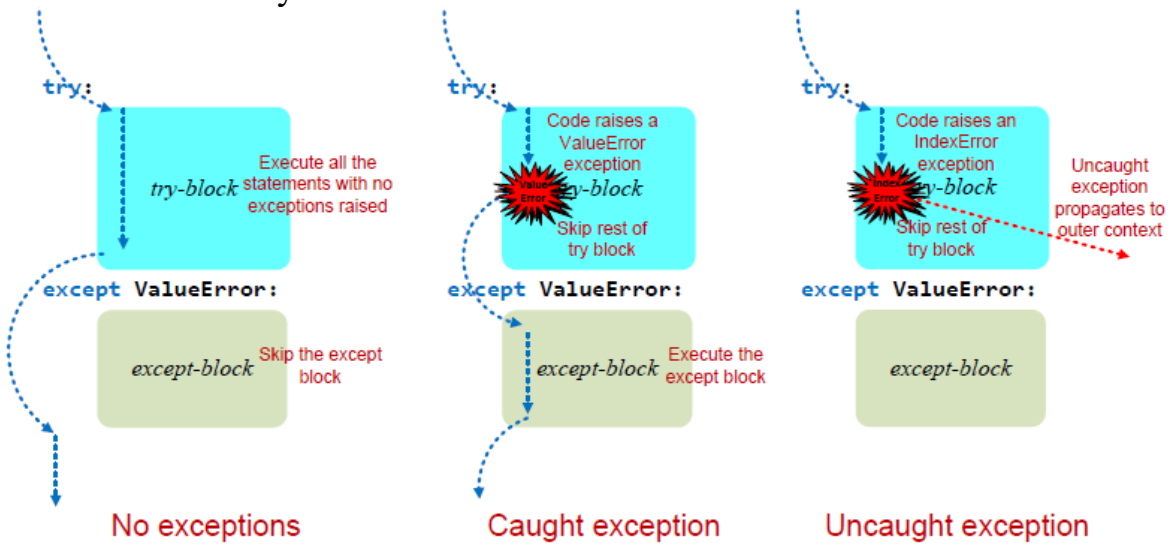
1.2.1. Інструкція try ... except ... else ... finally

Для обробки винятків призначена інструкція *try*. Формат інструкції:

```
try:
    <Блок, в якому перехоплюються винятки>
[except [<Виняток1> [as <Об'єкт винятку>]]:
    <Блок, що виконується при виникненні винятку>
[...
except [<ВинятокN> [as <Об'єкт виняток>]]:
    <Блок, що виконується при виникненні винятку>]]
```

```
[else:
    <Блок, що виконується якщо виняток не виникнув>]
[finally:
    <Блок, що виконується в будь-якому випадку>]
```

Інструкції, в яких перехоплюються винятки, повинні бути розташовані всередині блоку *try*. У блоці *except* в параметрі *<Виняток1>* вказується клас оброблюваного винятку.



Наприклад, обробити виняток, що виникає при діленні на нуль, можна так:

```
try:                                # Перехоплюємо винятки
    x = 1/0                         # Помилка: ділення на 0
except ZeroDivisionError:           # Вказуємо клас винятку
    print("Обробили ділення на 0")
    x = 0
print(x)                            # Виведе: 0
```

Результат:

```
Обробили ділення на 0
0
```

Якщо в блоці *try* виник виняток, управління передається блоку *except*. У разі якщо виняток не відповідає зазначеному класу, управління передається наступному блоку *except*. Якщо жоден блок *except* не відповідає винятку, то виняток «спливає» до обробника більш високого рівня. Якщо виняток в програмі взагалі ніде не обробляється, він передається обробнику за замовчуванням, який зупиняє виконання програми і виводить стандартну інформацію про помилку. Таким чином, в обробнику може бути присутнім кілька блоків *except* з різними класами винятків.

Крім того, один обробник можна вкласти в інший:

```
try:                                # Обробляємо винятки
    try:                            # Вкладений обробник
        x = 1/0                     # Помилка: ділення на 0
    except NameError:
        print("Невизначений ідентифікатор")
    except IndexError:
        print("Неіснуючий індекс")
    print("Вираз після вкладеного обробника")
except ZeroDivisionError:
    print("Обробка ділення на 0")
```

```
    x = 0
print(x)                              # Виведе: 0
```

У цьому прикладі у вкладеному обробнику не вказано виняток *ZeroDivisionError*, тому виняток «спливає» до обробника більш високого рівня. Після обробки винятку управління передається інструкції, розташованій відразу після обробника. У нашому прикладі управління буде передано інструкції, що виводить значення змінної *x* - *print(x)*. Зверніть увагу на те, що інструкція *print("Вираз після вкладеного обробника")* виконана не буде.

В інструкції *except* можна вказати відразу кілька винятків, записавши їх через кому всередині круглих дужок:

```
try:
    x = 1 / 0
except (NameError, IndexError, ZeroDivisionError):
    # Обробка відразу декількох винятків
    x = 0
print(x) # Виведе: 0
```

Отримати інформацію який виняток обробляється можна через другий параметр в інструкції *except*:

```
try:
    x = 1/0                           # Помилка: ділення на 0
except(NameError, IndexError, ZeroDivisionError) as err:
    print(err.__class__.__name__)     # Назва класу винятку
    print(err)                        # Текст повідомлення про помилку
```

Результат виконання:

```
ZeroDivisionError
division by zero
```

Перехоплення винятків використовується при написанні функцій:

```
def list_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
    ## ValueError: value is not in list
        index = -1
    return index

print(list_find([3,5,6,7], -6)) # Виведе: -1
```

Якщо в інструкції *except* не вказано клас винятку, то такий блок буде перехоплювати всі винятки. Приклад порожньої інструкції *except*:

```
try:
    x = 1/0 # Помилка: ділення на 0
except:     # Обробка всіх винятків
    x = 0
print(x)    # Виведе: 0
```

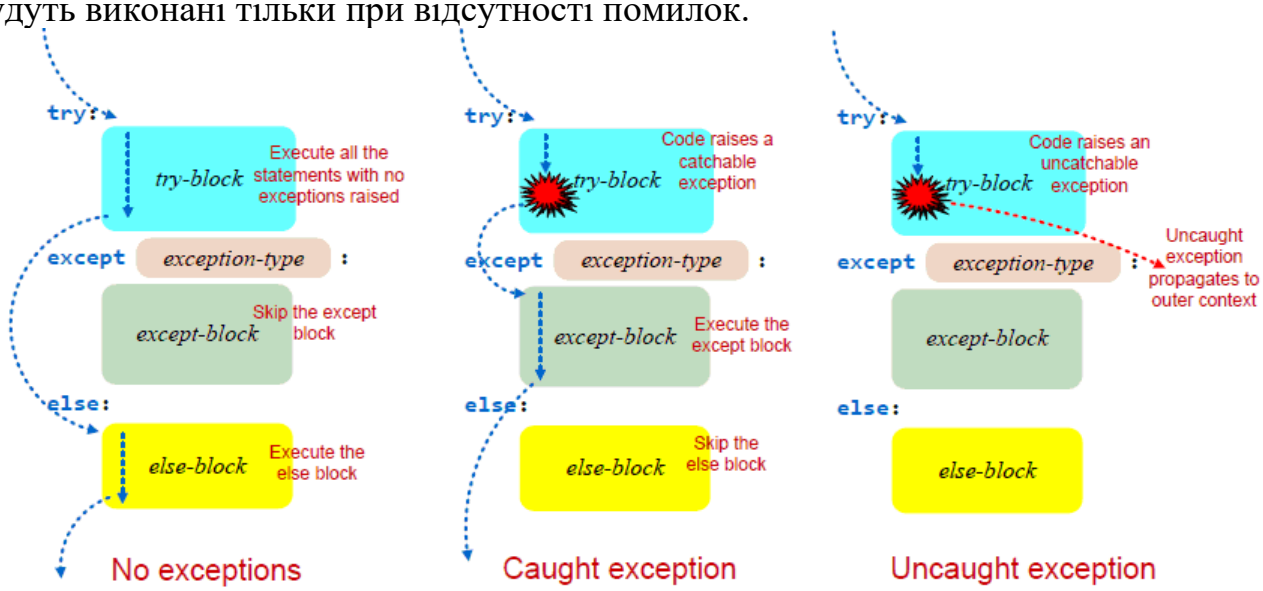
На практиці слід уникати порожніх інструкцій *except*, оскільки можна перехопити виняток, який є лише сигналом системі, а не помилкою. Наприклад, в наступному фрагменті коду відключається переривання зацикленої програми з допомогою комбінації клавіш *<Ctrl>+<C>*, які генерують виняток *KeyboardInterrupt*:

```
from random import randrange
```

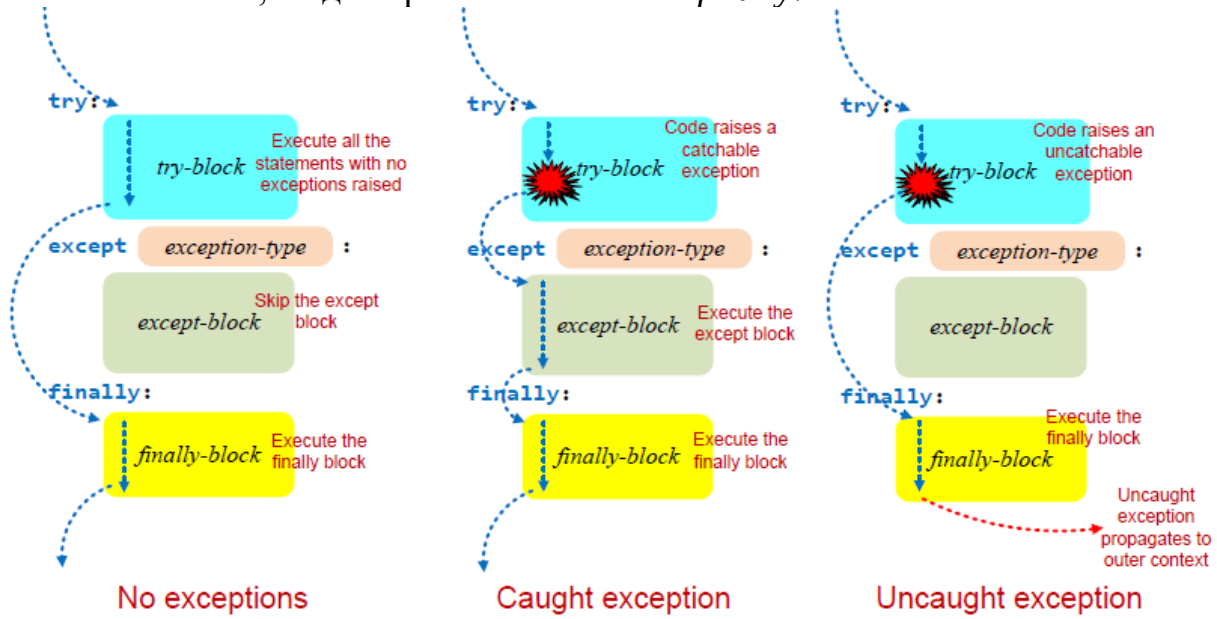
```
def main():
    number = randrange(100)
    while True:
        try:
            guess = int(input("? "))
        except:
            continue
        if guess == number:
            print("Ви виграли!")
            break

if __name__ == '__main__':
    main()
```

Якщо в обробнику присутній блок *else*, то інструкції всередині цього блоку будуть виконані тільки при відсутності помилок.



При необхідності виконати будь-які завершальні дії незалежно від того, виник виняток чи ні, слід скористатися блоком *finally*.



Для прикладу виведемо послідовність виконання блоків:

```
try:
    x = 10/2
    # x = 10/0
```

```
except ZeroDivisionError:
    print("Ділення на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Результат виконання при відсутності винятку:

```
Блок else
Блок finally
```

Послідовність виконання блоків при наявності винятку буде інша:

```
Ділення на 0
Блок finally
```

Необхідно зауважити, що при наявності винятку і відсутності блоку *except* інструкції всередині блоку *finally* будуть виконані, але виняток не буде оброблено. Він продовжить «спливання» до обробника більш високого рівня. Якщо користувальницький обробник відсутній, управління передається обробнику за замовчуванням, який перериває виконання програми і виводить повідомлення про помилку:

```
try:
    x = 10/0
finally: print("Блок finally")
```

Результат:

```
Блок finally
Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v1.py", line 86, in <module>
    x = 10/0
ZeroDivisionError: division by zero
```

Інструкція обробки винятків має декілька додаткових можливостей. Розглянемо їх на прикладі:

```
try:
    x = int(input("Введіть число: "))
    print(5/x)
except ZeroDivisionError as z:
    print("Обробляємо виняток - ділення на ноль!")
    print(z) # Виводимо інформацію про виняток ZeroDivisionError
except ValueError as v:
    print("Обробляємо виняток - перетворення типів!")
    print(v)
else:
    print("Виконується, якщо не було винятків!")
finally:
    print("Виконується завжди і в останню чергу!")
```

Результат:

```
Введіть число: 3
1.6666666666666667
Виконується, якщо не було винятків!
Виконується завжди і в останню чергу!
>>>
Введіть число: 0
Обробляємо виняток - ділення на ноль!
division by zero
Виконується завжди і в останню чергу!
```

```
>>>
Введіть число: d
Обробляємо виняток - перетворення типів!
invalid literal for int() with base 10: 'd'
Виконується завжди і в останню чергу!
```

Як приклад переробимо нашу програму підсумовування довільної кількості цілих чисел, введених користувачем, таким чином, щоб при введенні рядка замість числа програма не завершувалася з фатальною помилкою:

```
print("Введіть слово 'stop' для отримання результату")
summa = 0
while True:
    x = input( "Введіть число: ")
    if x == "stop":
        break # Вихід з циклу
    try:
        x = int(x) # Перетворимо рядок в число
    except ValueError:
        print("Необхідно ввести ціле число!")
    else:
        summa += x
print("Сума чисел рівна:", summa)
input()
```

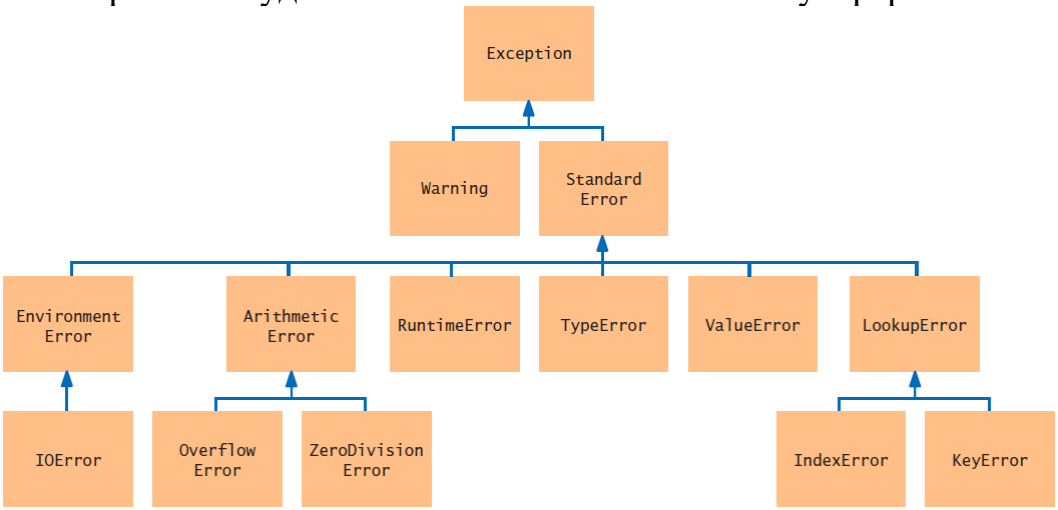
Процес введення значень і отримання результату виглядає так:

```
Введіть слово 'stop' для отримання результату
Введіть число: 1
Введіть число: w
Необхідно ввести ціле число!
Введіть число: 3
Введіть число: stop
Сума чисел рівна: 4
```

1.2.2. Класи вбудованих винятків

Усі вбудовані винятки в мові Python є класами. Python має досить розвинуту ієрархію вбудованих класів винятків. Вони дають змогу точно передавати суть помилки в *try-except* захищеному коді.

Найпоширеніші вбудовані класи винятків мають таку ієрархію.



Основна перевага використання класів для обробки винятків полягає в

можливості вказання базового класу для перехоплення всіх винятків відповідних похідних класів. Наприклад, для перехоплення ділення на нуль ми використовували клас *ZeroDivisionError*, але якщо замість нього вказати базовий клас *ArithmeticError*, перехоплюватимуться винятки класів *FloatingPointError, OverflowError* і *ZeroDivisionError*:

```
try:
    x = 1/0             # Помилка! Ділення на 0
except ArithmeticError: # Вказуємо базовий клас
    print("Обробили ділення на 0")
```

Розглянемо основні класи вбудованих винятків:

♦ *BaseException* - є класом самого верхнього рівня і базовим для всіх інших класів винятків;

♦ *Exception* - базовий клас для більшості вбудованих в Python винятків. Саме його, а не *BaseException* необхідно наслідувати при створенні користувацького класу винятку;

♦ *AssertionError* - генерується інструкцією *assert*;

♦ *AttributeError* - спроба звернення до неіснуючого атрибуту об'єкту;

♦ *EOFError* - генерується функцією *input()* при досягненні кінця файлу;

♦ *ImportError* - неможливо імпортувати модуль або пакет;

♦ *IndexError* - зазначений індекс не існує в послідовності;

♦ *KeyError* - зазначений ключ не існує в словнику;

♦ *KeyboardInterrupt* - натиснута комбінація клавіш <*Ctrl*>+<*C*>;

♦ *MemoryError* - інтерпретатору істотно не вистачає оперативної пам'яті;

♦ *NameError* - спроба звернення до ідентифікатора до його визначення;

♦ *OSError* - базовий клас для всіх винятків, генерованих при виникненні помилок в ОС (відсутність запитаного файлу, недостача місця на диску тощо.);

♦ *OverflowError* - число, що вийшло в результаті виконання арифметичної операції, занадто велике, щоб Python зміг його обробити;

♦ *RuntimeError* - некласифіковані помилка часу виконання;

♦ *StopIteration* - генерується методом *__next__()* як сигнал про закінчення ітерацій;

♦ *SyntaxError* - синтаксична помилка;

♦ *SystemError* - помилка в самій програмі інтерпретатора Python;

♦ *TabError* - в вихідному коді програми зустрівся символ табуляції, використання якого для створення відступів неприпустимо;

♦ *TypeError* - тип об'єкта не відповідає очікуваному;

♦ *UnboundLocalError* - всередині функції змінній присвоюється значення після звернення до однойменної глобальної змінної;

♦ *UnicodeDecodeError* - помилка перетворення послідовності байтів в рядок;

♦ *UnicodeEncodeError* - помилка перетворення рядка в послідовність байтів;

♦ *UnicodeTranslationError* - помилка перетворення рядка в іншу систему кодування;

♦ *ValueError* - переданий параметр не відповідає очікуваному значенню;

♦ *ZeroDivisionError* - спроба ділення на нуль.

Прослідкувати ієрархію для конкретного класу винятку можна з допомогою атрибуту *__mro__* – method resolution order:

```
>>> IndexError.__mro__
(<class 'IndexError'>, <class 'LookupError'>, <class 'Exception'>,
<class 'BaseException'>, <class 'object'>)
```

Наприклад:

```
>>> s =[1, 4, 6]; s[5]
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    s =[1, 4, 6]; s[5]
IndexError: list index out of range
>>> d = dict(a = 1, b = 2, c = 3); d['x']
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    d = dict(a = 1, b = 2, c = 3); d['x']
KeyError: 'x'
```

1.2.3. Користувацькі винятки

Для генерування користувацьких винятків призначені дві інструкції: *raise* і *assert*. Інструкція *raise* генерує заданий виняток. Вона має кілька варіантів формату:

```
raise <Екземпляр класу>
raise <Назва класу>
raise <Екземпляр або назва класу> from <Об'єкт>
raise
```

У *першому варіанті* формату інструкції *raise* вказується екземпляр класу генерованого винятку. При створенні екземпляра можна передати конструктору класу дані, які стануть доступні через другий параметр в інструкції *except*. Наведемо приклад генерації вбудованого винятку *ValueError*:

```
>>> raise ValueError("Опис винятку")
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    raise ValueError("Опис винятку")
ValueError: Опис винятку
```

Приклад обробки цього винятку:

```
try:
    raise ValueError("Опис винятку")
except ValueError as msg:
    print(msg) # Виведе: Опис винятку
```

Як виняток можна вказати екземпляр користувацького класу:

```
class MyError(Exception):
    def __init__ (self, value):
        self.msg = value
    def __str__ (self):
        return self.msg

# Обробка користувацького винятку
try:
    raise MyError("Опис винятку")
except MyError as err:
    print(err)     # Викликається метод __str__()
    print(err.msg) # Звернення до атрибуту класу
# Повторно генеруємо виняток
```

```
raise MyError("Опис винятку")
```

Результат виконання:

```
Опис винятку
Опис винятку
Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v1.py", line 136, in <module>
    raise MyError("Опис винятку")
MyError: Опис винятку
```

Клас *Exception* підтримує всі необхідні методи для виведення повідомлення про помилку. Тому в більшості випадків досить створити порожній клас, який успадковує клас *Exception*:

```
class MyError(Exception):
    pass

try:
    raise MyError("Опис винятку")
except MyError as err:
    print(err) # Виведе: Опис винятку
```

Для створення класу можна використати похідний від *Exception* клас:

```
import math

class NegativeNumberError(ValueError):
    """Attempted improper operation on negative number."""
    pass

def squareRoot(number):
    """Raises NegativeNumberError if number is less than 0."""
    if number < 0:
        raise NegativeNumberError("Square root of negative number not
permitted")

    return math.sqrt(number)

if __name__ == "__main__":
    squareRoot(-3)
```

У *другому варіанті* формату інструкції *raise* в першому параметрі задається об'єкт класу, а не екземпляр:

```
try:
     raise ValueError # Еквівалентно: raise ValueError()
except ValueError:
     print("Повідомлення про помилку")
```

У *третьому варіанті* формату інструкції *raise* в першому параметрі задається екземпляр класу або просто назва класу, а в другому параметрі вказується об'єкт винятку. У цьому випадку об'єкт винятку зберігається в атрибуті *__cause__*. При обробці вкладених винятків ці дані використовуються для виведення інформації не тільки про останній виняток, але і про початковий виняток. Приклад цього варіанта формату інструкції *raise*:

```
try:
    x = 1 / 0
except Exception as err:
    raise ValueError() from err
```

Результат виконання:

```
Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v1.py", line 155, 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 "D:\Python37\Work\lecture_7_v1.py", line 157, in <module>
    raise ValueError() from err
ValueError
```

Як видно з результату, ми отримали інформацію не тільки по винятку *ValueError*, але і по винятку *ZeroDivisionError*. Слід зауважити, що при відсутності інструкції *from* інформація зберігається неявним чином. Якщо прибрати інструкцію *from* в попередньому прикладі, ми отримаємо наступний результат:

```
Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v1.py", line 155, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v1.py", line 157, in <module>
    raise ValueError() #from err
ValueError
```

Четвертий варіант формату інструкції *raise* дозволяє повторно згенерувати останній виняток і зазвичай застосовується в коді, наступному за інструкцією *except*. Приклад цього варіанту:

```
class MyError(Exception):
    pass

try:
    raise MyError( "Повідомлення про помилку")
except MyError as err:
    print(err)
    raise          # Повторно генеруємо виняток
```

Результат виконання:

```
Повідомлення про помилку
Traceback (most recent call last):
  File "D:\Python37\Work\lecture_7_v1.py", line 164, in <module>
    raise MyError( "Повідомлення про помилку")
MyError: Повідомлення про помилку
```

Якщо виняток створюється з декількома аргументами ці аргументи передаються обробнику у формі кортежу, до якого можна звернутися через змінну *args* об’єкту помилки

```
class MyError(Exception):
    pass

try:
    raise MyError("Інформація про помилку", "my_filename", 3)
```

```
except MyError as error:
    print("Ситуація:    {0}    з    файлом    {1}\n"    "Error    code:
{2}".format(error.args[0], error.args[1], error.args[2]))
```

Результат:

```
Ситуація: Інформація про помилку з файлом my_filename
Error code: 3
```

Інструкція *assert* генерує виняток *AssertionError*, якщо логічний вираз повертає значення *False*. Інструкція має такий вигляд:

```
assert <Логічний вираз> [,<Дані>]
```

Наприклад:

```
>>> assert False, "The condition was false."
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    assert False, "The condition was false."
AssertionError: The condition was false.
>>> assert True, "The condition was true." # Нічого не відбудеться
```

Інструкція *assert* еквівалентна наступному коду:

```
if __debug__:
    if not <Логічний вираз>:
        raise AssertionError(<Дані>)
```

Якщо при запуску програми використовується прапорець *-O*, то змінна *__debug__* буде мати значення *False*. Так можна видалити всі інструкції *assert* з байт-коду. Таким чином, з допомогою команд *assert* можна облаштувати код командами відлагоджувального виводу на стадії розробки і залишити їх в коді на майбутнє без жодних витрат ресурсів при нормальному використанні.

Приклад використання інструкції *assert*:

```
try:
    x = -3
    assert x >= 0, "Повідомлення про помилку"
except AssertionError as err:
    print(err) # Виведе: Повідомлення про помилку
```

Пам’ятайте, інструкція *assert* – це скоріше засіб відладки, а не механізм обробки помилок часу виконання програми.

2. ЗАВДАННЯ

2.1. Домашня підготовка до роботи

1. Вивчити теоретичний матеріал.

2.2. Виконати в лабораторії

1. Задано каталог folder_1 з підкаталогами folder_2 і folder_3 (каталог є в матеріалах до лабораторної роботи). Написати програму маніпулювання з файлами і каталогами відповідно до завдання в табл. 1.

   Операції:

   1. Вивести вміст каталогу folder_1 і його підкаталогів.
   2. Створити в каталогу folder_1 підкаталог folder_4.

3. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.txt.
4. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.doc.
5. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.docx.
6. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.pdf.
7. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.xlsx.
8. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.txt.
9. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.doc.
10. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.docx.
11. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.pdf.
12. Скопіювати в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.xlsx.
13. Перемістити в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.txt.
14. Перемістити в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.doc.
15. Перемістити в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.docx.
16. Перемістити в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.pdf.
17. Перемістити в підкаталог folder_4 всі файли з каталогу folder_2 з розширенням *.xlsx.
18. Перемістити в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.txt.
19. Перемістити в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.doc.
20. Перемістити в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.docx.
21. Перемістити в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.pdf.
22. Перемістити в підкаталог folder_4 всі файли з каталогу folder_3 з розширенням *.xlsx.

23. Перейменувати в каталогу folder_1 всі файли з розширенням *.txt в ім’я файлу_2024.txt (напр., файл *file_1.txt* має стати *file_1_2024.txt*)
24. Перейменувати в каталогу folder_1 всі файли з розширенням *.doc в ім’я файлу_2024.doc (напр., файл *file_1.doc* має стати *file_1_2024.doc*)
25. Перейменувати в каталогу folder_1 всі файли з розширенням *.docx в ім’я файлу_2024.docx (напр., файл *file_1.docx* має стати *file_1_2024.docx*)
26. Перейменувати в каталогу folder_1 всі файли з розширенням *.pdf в ім’я файлу_2024.pdf (напр., файл *file_1.pdf* має стати *file_1_2024.pdf*)
27. Перейменувати в каталогу folder_1 всі файли з розширенням *.xlsx в ім’я файлу_2024. xlsx (напр., файл *file_1.xlsx* має стати *file_1_2024.xlsx*)
28. Перейменувати в підкаталогу folder_2 всі файли з розширенням *.txt в ім’я файлу_2024.txt (напр., файл *file_1.txt* має стати *file_1_2024.txt*)
29. Перейменувати в підкаталогу folder_2 всі файли з розширенням *.doc в ім’я файлу_2024.doc (напр., файл *file_1.doc* має стати *file_1_2024.doc*)
30. Перейменувати в підкаталогу folder_2 всі файли з розширенням *.docx в ім’я файлу_2024.docx (напр., файл *file_1.docx* має стати *file_1_2024.docx*)
31. Перейменувати в підкаталогу folder_2 всі файли з розширенням *.pdf в ім’я файлу_2024.pdf (напр., файл *file_1.pdf* має стати *file_1_2024.pdf*)
32. Перейменувати в підкаталогу folder_2 всі файли з розширенням *.xlsx в ім’я файлу_2024. xlsx (напр., файл *file_1.xlsx* має стати *file_1_2024.xlsx*)
33. Перейменувати в підкаталогу folder_3 всі файли з розширенням *.txt в ім’я файлу_2024.txt (напр., файл *file_1.txt* має стати *file_1_2024.txt*)
34. Перейменувати в підкаталогу folder_3 всі файли з розширенням *.doc в ім’я файлу_2024.doc (напр., файл *file_1.doc* має стати *file_1_2024.doc*)
35. Перейменувати в підкаталогу folder_3 всі файли з розширенням *.docx в ім’я файлу_2024.docx (напр., файл *file_1.docx* має стати *file_1_2024.docx*)
36. Перейменувати в підкаталогу folder_3 всі файли з розширенням *.pdf в ім’я файлу_2024.pdf (напр., файл *file_1.pdf* має стати *file_1_2024.pdf*)
37. Перейменувати в підкаталогу folder_3 всі файли з розширенням *.xlsx в ім’я файлу_2024. xlsx (напр., файл *file_1.xlsx* має стати *file_1_2024.xlsx*)
38. Видалити в підкаталогу folder_1 всі файли з розміром більше 100 КБайт.
39. Видалити в підкаталогу folder_1 всі файли з розширенням .txt.
40. Видалити в підкаталогу folder_1 всі файли з розширенням doc.
41. Видалити в підкаталогу folder_1 всі файли з розширенням .docx.
42. Видалити в підкаталогу folder_1 всі файли з розширенням .pdf.
43. Видалити в підкаталогу folder_1 всі файли з розширенням .xlsx.
44. Видалити в підкаталогу folder_2 всі файли з розміром більше 100 КБайт.
45. Видалити в підкаталогу folder_2 всі файли з розширенням .txt.
46. Видалити в підкаталогу folder_2 всі файли з розширенням doc.

47. Видалити в підкаталогу folder_2 всі файли з розширенням .docx.
48. Видалити в підкаталогу folder_2 всі файли з розширенням .pdf.
49. Видалити в підкаталогу folder_2 всі файли з розширенням .xlsx.
50. Видалити в підкаталогу folder_3 всі файли з розміром більше 100 КБайт.
51. Видалити в підкаталогу folder_3 всі файли з розширенням .txt.
52. Видалити в підкаталогу folder_3 всі файли з розширенням doc.
53. Видалити в підкаталогу folder_3 всі файли з розширенням .docx.
54. Видалити в підкаталогу folder_3 всі файли з розширенням .pdf.
55. Видалити в підкаталогу folder_3 всі файли з розширенням .xlsx.
56. Видалити підкаталог folder_2.
57. Видалити підкаталог folder_3.

Табл. 1

Варіанти завдань

Варіант	**Операції**
1.	1, 2, 3, 14, 23, 38, 56
2.	1, 2, 4, 13, 24, 39, 57
3.	1, 2, 5, 16, 25, 40, 56
4.	1, 2, 6, 15, 26, 41, 57
5.	1, 2, 7, 18, 27, 42, 56
6.	1, 2, 8, 17, 28, 43, 57
7.	1, 2, 9, 20, 29, 44, 56
8.	1, 2, 10, 19, 30, 45, 57
9.	1, 2, 12, 21, 31, 46, 56
10.	1, 2, 11, 22, 32, 47, 57
11.	1, 2, 8, 16, 33, 49, 56
12.	1, 2, 7, 18, 34, 48, 57
13.	1, 2, 9, 17, 35, 50, 56
14.	1, 2, 10, 19, 36, 51, 57
15.	1, 2, 11, 20, 37, 52, 56
16.	1, 2, 12, 21, 27, 53, 57
17.	1, 2, 3, 22, 28, 54, 56
18.	1, 2, 4, 13, 29, 55, 57
19.	1, 2, 5, 14, 30, 43, 56
20.	1, 2, 6, 15, 31, 44, 57
21.	1, 2, 10, 19, 32, 45, 56
22.	1, 2, 11, 20, 33, 46, 57
23.	1, 2, 12, 21, 34, 47, 56
24.	1, 2, 3, 22, 35, 48, 57
25.	1, 2, 4, 16, 36, 49, 56
26.	1, 2, 5, 17, 37, 50, 57
27.	1, 2, 6, 18, 23, 52, 56
28.	1, 2, 7, 13, 24, 52, 57
29.	1, 2, 8, 14, 25, 53, 56
30.	1, 2, 9, 15, 26, 54, 57

2. Програму з лабораторної роботи № 7 доповнити таким чином, щоб список при запуску програми завантажувався з текстового файлу, а при виході зберігався у текстовий файл. Додати пункти меню і відповідні функції для таких операцій: “Зберегти список у текстовий файл”, “Зберегти список у файл як об’єкт”, “Завантажити список з текстового файлу”, “Завантажити список як об’єкт”. Також доповнити програму обробкою винятків, що виникають при неправильно введених даних, неправильно заданих індексах елементів, файлових помилках. Програма повинна коректно працювати при будь-яких діях користувача.

3. Програму валідації введеного паролю з Лабораторної роботи №7 доповнити додатковою перевіркою пароля: перевіряється чи є пароль у заданому текстовому файлі згідно табл. 2 – якщо так, то виводиться повідомлення, що пароль слабкий і валідація не проходить. Файли з паролями є в матеріалах до лабораторної роботи.

Табл. 2

4. Варіанти завдань

Варіант	Операції
1.	10-million-password-list-top-10000.txt
2.	10-million-password-list-top-100000.txt
3.	10-million-password-list-top-1000000.txt
4.	100k-most-used-passwords-NCSC.txt
5.	10-million-password-list-top-10000.txt
6.	10-million-password-list-top-100000.txt
7.	10-million-password-list-top-1000000.txt
8.	100k-most-used-passwords-NCSC.txt
9.	10-million-password-list-top-10000.txt
10.	10-million-password-list-top-100000.txt
11.	10-million-password-list-top-1000000.txt
12.	100k-most-used-passwords-NCSC.txt
13.	10-million-password-list-top-10000.txt
14.	10-million-password-list-top-100000.txt
15.	10-million-password-list-top-1000000.txt
16.	100k-most-used-passwords-NCSC.txt
17.	10-million-password-list-top-10000.txt
18.	10-million-password-list-top-100000.txt
19.	10-million-password-list-top-1000000.txt
20.	100k-most-used-passwords-NCSC.txt
21.	10-million-password-list-top-10000.txt
22.	10-million-password-list-top-100000.txt
23.	10-million-password-list-top-1000000.txt
24.	100k-most-used-passwords-NCSC.txt
25.	10-million-password-list-top-10000.txt
26.	10-million-password-list-top-100000.txt
27.	10-million-password-list-top-1000000.txt
28.	100k-most-used-passwords-NCSC.txt

29.	10-million-password-list-top-10000.txt
30.	10-million-password-list-top-100000.txt

3. ЗМІСТ ЗВІТУ

1. Мета роботи.
2. Повний текст завдання згідно варіанту.
3. Лістинг програми.
4. Результати роботи програм (у текстовій формі та скріншот).
5. Висновок.

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. В чому різниця між текстовим файлом і бінарним?
2. Якщо програма виконує *raise*, яка інструкція буде виконаною наступною?
3. Коли буде виконуватися блок *finally* в інструкції *try*?

5. СПИСОК ЛІТЕРАТУРИ

1. Learn to Program with Python 3. A Step-by-Step Guide to Programming, Second Edition / Irv Kalb. – Mountain View: Apress, 2018. – 361 p.
2. The Python Workbook. A Brief Introduction with Exercises and Solutions, Second Edition / Ben Stephenson. – Cham: Springer, 2014. – 218 p.
3. Python Pocket Reference, Fifth Edition / Mark Lutz. – Sebastopol: O'Reilly Media, Inc., 2014. – 264 p.
4. Learn Python 3 the Hard Way / Zed A. Shaw. – Boston: Addison-Wesley, 2017. – 321 p.
5. A Python Book: Beginning Python, Advanced Python, and Python Exercises / Dave Kuhlman. – Boston: MIT, 2013. – 278 p.

НАВЧАЛЬНЕ ВИДАННЯ

Створення та використання функцій

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 7
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Укладач: Я. Р. Совин, канд. техн. наук, доцент