МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра “Захист інформації”

Робота з рядками

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 3
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 рядками: Методичні вказівки до лабораторної роботи № 3 з курсу «Програмування скриптовими мовами» для студентів спеціальності «Кібербезпека» / Укл. *Я. Р. Совин* – Львів: Національний університет "Львівська політехніка", 2024. – 27 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 вбудованими рядковими типами Python та операторами і функціями для роботи з ними**.**

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Типи рядків

Рядки це послідовності символів. Довжина рядка обмежена лише об'ємом оперативної пам’яті комп’ютера. Як і всі послідовності, рядки підтримують звернення до елементу за індексом, отримання зрізу, конкатенацію (оператор +), повторення (оператор *), а також перевірку на входження (оператори *in* та *not in*). Крім того, рядки належать до незмінних типів даних. Тому практично всі рядкові методи в якості значення повертають новий рядок. (При використанні невеликих рядків це не призводить до будь-яких проблем, але при роботі з великими рядками можна зіткнутися з проблемою нестачі пам’яті.) Іншими словами, можна отримати символ за індексом, але змінити його буде не можливо:

```
>>> s = "Python"
>>> s[0] # Можна отримати символ за індексом
'P'
>>> s[0] = "J" # Помилка! Змінити рядок не можна
```

У деяких мовах програмування рядок повинен закінчуватися нульовим символом. У мові Python нульовий символ може бути розташований всередині рядка:

```
>>> "string\x00string" # Нульовий символ — це не кінець рядка
'string\x00string'
```

Python підтримує наступні рядкові типи:

♦ *str* – Unicode-рядок. Зверніть увагу, конкретне кодування: UTF-8, UTF-16 або UTF-32 – тут не вказане. Розглядайте такі рядки, як рядки в якомусь абстрактному кодуванні, що дозволяють зберігати символи Unicode і проводити маніпуляції з ними. При виведенні Unicode-рядок необхідно перетворити в послідовність байтів в будь-якому кодуванні:

```
>>> type("рядок")
<class 'str'>
>>> "рядок".encode(encoding = "cp1251")
b'\xf0\xff\xe4\xee\xea'
>>> "рядок".encode (encoding = "utf-8")
b'\xd1\x80\xd1\x8f\xd0\xb4\xd0\xbe\xd0\xba'
```

♦ *bytes* – незмінна послідовність байтів. Кожен елемент послідовності може зберігати ціле число від про до 255, яке позначає код символу. Об'єкт типу *bytes* підтримує більшість рядкових методів і, якщо це можливо, виводиться як послідовність символів. Однак доступ за індексом повертає ціле число, а не символ:

```
>>> s = bytes("рядок str", "cp1251")
>>> s
b'\xf0\xff\xe4\xee\xea str'
>>> type(s)
<class 'bytes'>
>>> s[0], s[7], s[0:5], s[6:9]
```

```
(240, 116, b'\xf0\xff\xe4\xee\xea', b'str')
```

Об’єкт типу *bytes* може містити як однобайтові, так і багатобайтові символи. Зверніть увагу на те, що функції і методи рядків некоректно працюють з мультибайтними кодуваннями, – наприклад, функція *len()* поверне кількість байтів, а не символів:

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

♦ *bytearray* – змінна послідовність байтів. Тип *bytearray* аналогічний типу *bytes*, але дозволяє змінювати елементи за індексом і містить додаткові методи, які дають можливість додавати і видаляти елементи:

```
>>> s = bytearray("str", "cp1251")
>>> s
bytearray(b'str')
>>> type(s)
<class 'bytearray'>
>>> s[0] = 49; s    # Можна змінити символ
bytearray(b'1tr')
>>> s.append(55); s # Можна добавити символ
bytearray(b'1tr7')
```

У всіх випадках, коли мова йде про текстові дані, слід використовувати тип *str*. Саме цей тип ми будемо називати словом «рядок». Типи *bytes* і *bytearray* слід задіяти для запису двійкових даних (наприклад, під час шифрування чи обробки зображень) і проміжного зберігання рядків.

1.2. Створення рядків

Створити рядок можна за допомогою таких дій:

♦ За допомогою функції *str([<Об'єкт> [, <Кодування> [, <Обробка помилок>]]])*. Якщо вказано тільки перший параметр, функція повертає рядкове представлення будь-якого об'єкта. Якщо параметри не задані взагалі, повертається порожній рядок:

```
>>> str(), str([ 1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
```

♦ Вказавши рядок між апострофами або подвійними лапками:

```
>>> 'рядок', "рядок", '"x": 5', "'x': 5"
('рядок', 'рядок', '"x": 5', "'x': 5")
```

У мові Python ніякої відмінності між рядком в апострофах і рядком в лапках немає. Якщо рядок містить лапки, то його краще помістити в апострофи, і навпаки:

```
>>> print("Never Say 'Never' Again")
Never Say 'Never' Again
>>> print('Never Say "Never" Again')
Never Say "Never" Again
>>> print("Never Say "Never" Again")
SyntaxError: invalid syntax
```

Всі спеціальні символи в таких рядках інтерпретуються - наприклад, послідовність символів *n* перетворюється в символ нового рядка.

```
>>> print('Рядок1\nРядок2')
Рядок1
Рядок2
>>> print("Рядок1\nРядок2")
Рядок1
Рядок2
```

Щоб спеціальний символ виводився як є, його необхідно екранувати за допомогою захисного слеша:

```
>>> print ("Рядок1\\nРядок2")
Рядок1\nРядок2
```

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

```
>>> "\"x\": 5", '\'x\': 5'
('"x": 5', "'x': 5")
```

Щоб розташувати об'єкт на декількох рядках, слід перед символом переводу рядка вказати символ \, помістити два рядки всередині дужок або використовувати конкатенацію всередині дужок:

```
>>> "stringl\
string2" # Після \ не повинно бути ніяких символів
'stringlstring2'
>>> ("stringl"
"string2") # Неявна конкатенація рядків
'stringlstring2'
>>> ("stringl" +
"string2") # Явна конкатенація рядків
'stringlstring2'
```

♦ Вказавши рядок між потроєними апострофами або потроєними лапками. такі об'єкти можна розмістити на декількох рядках. Допускається також одночасно використовувати і лапки, і апострофи без необхідності їх екранувати. В іншому такі об’єкти еквівалентні рядкам в апострофа і лапках. Всі спеціальні символи в таких рядках інтерпретуються:

```
>>> print("""Рядок1
Рядок2""")
Рядок1
Рядок2
```

Якщо рядок не присвоюється змінній, то він вважається рядком документування. Такий рядок зберігається в атрибуті *__doc__* того об'єкту, в якому розташований. В якості прикладу створимо функцію з рядком документування, а потім виведемо вміст рядка:

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

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

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

```
>>> print("Рядок1\nРядок2")
Рядок1
Рядок2
>>> print(r"Рядок1\nРядок2")
Рядок1\nРядок2
```

1.3. Спеціальні символи

Спеціальні символи – це комбінації знаків, що позначають службові або недруковані символи, які неможливо вставити звичайним способом. Наведемо перелік спеціальних символів, допустимих всередині рядка, перед яким немає модифікатора *r*:

- ♦ *n* – новий рядок;
- ♦ *r* – повернення каретки;
- ♦ *t* – знак табуляції;
- ♦ *v* – вертикальна табуляція;
- ♦ *a* – дзвінок;
- ♦ *b* – забій;
- ♦ *f* – перевід формату;
- ♦ *0* – нульовий символ (не є кінцем рядка);
- ♦ \" – лапки;
- ♦ \' – апостроф;
- ♦ *xN* – символ з шістнадцятковим кодом *N*. Наприклад, *x6a* відповідає символу *j*;
- ♦ *uxxxx* – 16-бітний символ Unicode. Наприклад, *u043a* відповідає букві *к*;
- ♦ *Uxxxxxxxx* – 32-бітний символ Unicode.

1.4. Операції над рядками

Рядки належать до послідовностей. Як і всі послідовності, рядки підтримують звернення до елементу за індексом, отримання зрізу, конкатенацію, повторення і перевірку на входження. Розглянемо ці операції детально.

До будь-якого символу рядка можна звернутися як до елементу списку – досить вказати його індекс в квадратних дужках. Нумерація починається з нуля:

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

В якості індексу можна вказати від'ємне значення. У цьому випадку зміщення буде відраховуватися від кінця рядка, а точніше – щоб отримати позитивний індекс, значення віднімається з довжини рядка:

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

Так як рядки належать до незмінних типів даних, то змінити символ за індексом не можна. Щоб виконати зміну, можна скористатися операцією вилучення зрізу, яка повертає зазначений фрагмент рядка. Формат операції:

```
[<Початок>:<Кінець>:<Крок>]
```

Всі параметри тут не є обов'язковими. Якщо параметр *<Початок>* не вказано, то використовується значення 0. Якщо параметр *<Кінець>* не вказано, то повертається фрагмент до кінця рядка. Слід також зауважити, що символ з індексом, зазначеним в цьому параметрі, не входить в фрагмент, що повертається. Якщо параметр *<Крок>* не вказано, то використовується значення 1. Як значення параметрів можна вказати негативні значення. Розглянемо кілька прикладів:

♦ спочатку отримаємо копію рядка:

```
>>> s = "Python"
>>> s[:] # Повертається фрагмент від позиції 0 до кінця рядка
'Python'
```

♦ тепер виведемо символи у зворотному порядку:

```
>>> s[::-1] # Вказуємо від'ємне значення в параметрі <Крок>
'nohtyP'
```

♦ замінимо перший символ в рядку:

```
>>> "J" + s [1:] # Витягуємо фрагмент від символу 1 до кінця рядка
'Jython'
```

♦ видалимо останній символ:

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

♦ отримаємо перший символ в рядку:

```
>>> s[0:1] # Символ з індексом 1 не входить в діапазон
'P'
```

♦ а тепер отримаємо останній символ:

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

♦ і, нарешті, виведемо символи з індексами 2, 3 і 4:

```
>>> s[2:5] # Повертаються символи з індексами 2, 3 і 4
'tho'
```

Дізнатися кількість символів в рядку (її довжину) дозволяє функція *len()*:

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

Тепер, коли ми знаємо кількість символів, можна перебрати всі символи за допомогою циклу *for*:

```
>>> s = "Python"
>>> for i in range(len(s)): print(s[i], end =" ")
P y t h o n
```

Так як рядки підтримують ітерації, ми можемо просто вказати рядок як параметр циклу:

```
>>> for i in s: print(i, end =" ")
P y t h o n
```

З'єднати два рядки в один рядок (виконати їх конкатенацію) дозволяє оператор +:

```
>>> print("рядок1" + "рядок2")
рядок1рядок2
>>> print("James" + " " + "Bond")
James Bond
```

Крім того, можна виконати неявну конкатенацію рядків. У цьому випадку два рядки вказуються поруч без оператора між ними:

```
>>> print("рядок1" "рядок2")
рядок1рядок2
```

Якщо з'єднуються, наприклад, змінна і рядок, то слід обов'язково вказувати символ конкатенації рядків, інакше буде виведено повідомлення про помилку:

```
>>> s = "рядок1"
>>> print(s + "рядок2") # Нормально
рядок1рядок2
```

При необхідності з'єднати рядок зі значенням іншого типу (наприклад, з числом) слід провести явне перетворення типів за допомогою функції *str()*:

```
>>> "String" + str(10)
'String10'
```

Крім розглянутих операцій, рядки підтримують операцію повторення, перевірки на входження і невходження. Повторити рядок вказану кількість разів можна за допомогою оператора *, виконати перевірку на входження фрагмента в рядок дозволяє оператор *in*, а перевірити на невходження – оператор *not in*:

```
>>> "-" * 20
'--------------------'
>>> "yt" in "Python"      # Знайдено
True
>>> "yt" in "Perl"        # Не знайдено
False
>>> "PHP" not in "Python" # Не знайдено
True
```

1.5. Форматування рядків

Замість з'єднання рядків за допомогою оператора + краще використовувати форматування. Ця операція дозволяє з'єднувати рядки зі значеннями будь-яких інших типів і виконується швидше конкатенації.

1.5.1. Форматування з допомогою оператора %

Оператор *%* використовується для об'єднання значень Python в відформатовані рядки для друку чи іншого застосування. Застосування *%* для форматування рядків відноситься до старого стилю форматування. У наступних версіях Python оператор форматування *%* може бути видалений. Замість цього оператора в новому коді слід використовувати метод *format()*, який розглядається далі.

Операція форматування записується в такий спосіб:

```
<Рядок спеціального формату> % <Значення>
```

Усередині параметра *<Рядок спеціального формату>* можуть бути вказані специфікатори, які мають наступний синтаксис:

```
%[(<Ключ>)][<Прапорець>][<Ширина>][.Точність>]<Тип перетворення>
```

Кількість специфікаторів всередині рядка має дорівнювати кількості елементів в параметрі *<Значення>*. Якщо використовується тільки один специфікатор, то параметр *<Значення>* може містити одне значення, в іншому випадку необхідно вказати значення через кому всередині круглих дужок:

```
>>> "%s" % 10 # Один елемент
'10'
>>> "My name is %s, %s %s" % ("Bond", "James", "Bond")
'My name is Bond, James Bond'
>>> "%s – %s – %s" % (10, 20, 30)
'10 – 20 – 30'
```

Параметри всередині специфікатора мають наступний сенс:

♦ *<Прапорець>* - прапорець перетворення. Може містити наступні значення:

• # – для шістнадцяткових значень додає в початок комбінацію символів *0x* (якщо використовується тип *x*) або *0X* (якщо використовується тип *X*), для дійсних чисел наказує завжди виводити дробову крапку, навіть якщо задано значення 0 в параметрі *<Точність>*.

```
>>> print("%#x %#x %#X %#X" % (0xff, 10, 0xff, 10))
0xff 0xa 0XFF 0XA
>>> print("%#.0F %#.0F" % (300, 300))
300. 300.
```

• *0* – задає наявність нулів на початку для числового значення:

```
>>> print("%d – %05d" % (3, 3)) # 5 – ширина поля
3 – 00003
>>> print("James Bond - Agent %03d" % (7))
James Bond - Agent 007
```

• - – задає вирівнювання по лівій межі області. За замовчуванням використовується вирівняний по правій границі. Якщо прапорець вказано одночасно з прапорцем 0, то дію прапорця 0 буде скасовано:

```
>>> "%5d - %-5d" % (3, 3) # 5 – ширина поля
'    3 - 3    '
>>> "'%05d' - '%-05d' " % (3, 3)
"'00003' - '3    ' "
```

• *пробіл* – вставляє пропуск перед позитивним числом. Перед негативним числом буде стояти мінус:

```
>>> "'% d' - '% d'" % (-3, 3)
"'-3' - ' 3'"
```

• + – задає обов'язковий вивід знаку як для негативних, так і для позитивних чисел. Якщо прапорець + вказано одночасно з прапорцем пробіл, то дію прапорця пробіл буде скасовано:

```
>>> "'%+d' - '%+d'" % (-3, 3)
"'-3' - '+3'"
```

♦ *<Ширина>* – мінімальна ширина поля. Якщо рядок не поміщається в зазначену ширину, значення ігнорується, і рядок виводиться повністю:

```
>>> "'%10d' - '%-10d'" % (3, 3)
"'         3' - '3         '"
>>> "'%3s' '%10s'" % ("string", "string")
"'string' '    string'"
```

♦ *<Точність>* – кількість знаків після коми для дійсних чисел. Перед цим параметром обов'язково повинна стояти крапка:

```
>>> "%s %f %.2f" % (math.pi, math.pi, math.pi)
'3.141592653589793 3.141593 3.14'
```

♦ *<Тип перетворення>* – задає тип перетворення. Параметр є обов'язковим. У параметрі *<Тип перетворення>* можуть бути вказані такі символи:

• *s* – перетворює будь-який об'єкт в рядок за допомогою функції *str()*:

```
>>> print("%s" % ("Звичайний рядок"))
Звичайний рядок
>>> print("%s %s %s" % (10, 10.52, [1, 2, 3]))
10 10.52 [1, 2, 3]
```

• *a* – перетворює об'єкт в рядок за допомогою функції *ascii()*:

```
>>> print("%a" % ("рядок string"))
'\u0440\u044f\u0434\u043e\u043a string'
```

• *c* – виводить одиночний символ або перетворює числове значення в символ. В якості прикладу виведемо числове значення і відповідний цьому значенню символ:

```
>>> for i in range(33, 127): print("%s => %c" % (i, i))
```

• *d* та *i* – повертають цілу частину числа:

```
>>> print("%d %d %d" % (10, 25.6, -80))
10 25 -80
>>> print("%i %i %i" % (10, 25.6, -80))
10 25 -80
```

• *x* – шістнадцяткове значення в нижньому регістрі:

```
>>> print("%x %x" % (0xff, 10))
ff a
>>> print("%#x %#x" % (0xff, 10))
0xff 0xa
```

• *X* – шістнадцяткове значення у верхньому регістрі:

```
>>> print("%X %X" % (0xff, 10))
FF A
>>> print("%#X %#X" % (0xff, 10))
0XFF 0XA
```

• *f* та *F* – дійсне число в десятковому поданні:

```
>>> print("%f %f %f" % (300, 18.65781452, -12.5))
300.000000 18.657815 -12.500000
```

• *e* та *E* – дійсне число в експоненційної формі:

```
>>> print("%e %e" % (3000, 18657.81452))
3.000000e+03 1.865781e+04
```

• *g* та *G* – еквівалентно f або e (вибирається більш короткий запис числа):

```
>>> print("%g %g %g" % (0.086578, 0.000086578, 1.865E-005))
0.086578 8.6578e-05 1.865e-05
```

Якщо всередині рядка необхідно використовувати символ відсотка, цей символ слід подвоїти, інакше буде виведено повідомлення про помилку:

```
>>> print("%% %s" % ("- це символ відсотку")) # Нормально
% - це символ відсотку
```

Швидкий тест на форматування рядків з допомогою оператора %. Що виведеться при виконанні таких рядків:

```
>>> x = "%.2f" % 1.1111; print(x)
>>> x = "%(a).2f" % {'a':1.1111}; print(x)
>>> x = "%(a).08f" % {'a':1.1111}; print(x)
```

Відповідь:

```
1.11
1.11
1.11110000
```

Для форматування рядків також можна використовувати такі методи:

♦ *expandtabs([<Ширина поля>])* – замінює символ табуляції пробілами таким чином, щоб загальна ширина фрагмента разом з текстом, розташованим

перед символом табуляції, дорівнювала зазначеній величині. Якщо параметр не вказано, то ширина поля передбачається рівною 8 символів:

```
>>> s = "1\tl2\tl23\t"
>>> "'%s'" % s.expandtabs(4)
"'1   l2  l23 '"
```

♦ *center(<Ширина>[,<Символ>])* – здійснює вирівнювання рядка по центру всередині поля зазначеної ширини. Якщо другий параметр не вказано, праворуч і ліворуч від вихідного рядка будуть додані пробіли:

```
>>> s = "str"
>>> s.center(15), s.center(11, "-")
('      str      ', '----str----')
```

♦ *ljust(<Ширина>[,<Символ>])* – здійснює вирівнювання рядка по лівому краю всередині поля зазначеної ширини. Якщо другий параметр не вказано, праворуч від вихідного рядка будуть додані пробіли. Якщо кількість символів в рядку перевищує ширину поля, значення ширини ігнорується, і рядок повертається повністю:

```
>>> s = "string"
>>> s.ljust(15), s.ljust(15, "-")
('string         ', 'string---------')
>>> s.ljust(6), s.ljust(5)
('string', 'string')
```

♦ *rjust(<Ширина>[, <Символ>])* – виробляє вирівнювання рядка по правому краю всередині поля зазначеної ширини. Якщо другий параметр не вказано, зліва від вихідного рядка будуть додані пробіли. Якщо кількість символів в рядку перевищує ширину поля, значення ширини ігнорується, і рядок повертається повністю:

```
>>> s = "string"
>>> s.rjust(15), s.rjust(15, "-")
('         string', '---------string')
>>> s.rjust(6), s.rjust(5)
('string', 'string')
```

♦ *zfill(<Ширина>)* – виробляє вирівнювання фрагменту по правому краю всередині поля зазначеної ширини. Зліва від фрагмента будуть додані нулі. Якщо кількість символівв рядку перевищує ширину поля, значення ширини ігнорується, і рядок повертається повністю:

```
>>> "5".zfill(20), "123456".zfill(5)
('00000000000000000005', '123456')
```

1.5.2. Форматування з допомогою методу format()

Новіший спосіб форматування, з ще більшою точністю і широтою можливостей, базується на використанні методу *format()* класу рядка. Команда *format* представляє собою потужну міні-мову форматування рядків майже з нескінченними можливостями. Зокрема з його допомогою можна задати символ-заповнювач, тип вирівнювання, знак, ширину, точність і тип даних і т.п.

Використання методу *format()* повинно виглядати так:

```
<Рядок> = <Рядок спеціального формату>.format(*args, **kwargs)
```

У параметрі *<Рядок спеціального формату>* всередині символів фігурних дужок *{}* вказуються специфікатори, що мають наступний синтаксис:

```
{[<Поле>][!<Функція>][:<Формат>]}
```

Всі символи, розташовані поза фігурними дужками, виводяться без перетворень. Якщо всередині рядка необхідно використовувати символи *{}*, то ці символи слід подвоїти, інакше отримаємо *ValueError*:

♦ Як параметр *<Поле>* можна вказати порядковий номер (нумерація починається з нуля) або ключ параметра, зазначеного в методі *format()*. Припустимо комбінувати позиційні та іменовані параметри, при цьому іменовані параметри слід вказати останніми:

```
>>> "{0} - {1} - {2}".format(10, 12.3, "string") # Індекси
'10 - 12.3 - string'
>>> "{model} - {color}".format(color = "red", model = "BMW") # Ключі
'BMW - red'
>>> "{color} - {0}".format(2015, color = "red") # Комбінація
'red - 2015'
```

Існує також коротка форма запису, при якій *<Поле>* не вказується. В цьому випадку душки без вказаного індексу нумеруються зліва направо, починаючи з нуля:

```
>>> "{} - {} - {} - {n}".format(1, 2, 3, n=4) # "{0} - {1} - {2} - {n}"
'1 - 2 - 3 - (n)'
>>> "{} - {} - {n} - {}".format(1, 2, 3, n=4) # "{0} - {1} - {n} - {2}"
'1 - 2 - 4 - 3'
```

Параметр *<Функція>* задає функцію, за допомогою якої обробляються дані перед вставкою в рядок. Якщо вказано значення *s*, то дані обробляються функцією *str()*, якщо значення *r*, то функцією *repr()*, а якщо значення *a*, то функцією *ascii()*. Якщо параметр не вказано, для перетворення даних в рядок використовується функція *str()*:

```
>>> print("{0!s}".format("рядок")) # str()
рядок
>>> print("{0!r}".format("рядок")) # repr()
'рядок'
>>> print("{0!a}".format("рядок")) # ascii()
'\u0440\u044f\u0434\u043e\u043a'
```

♦ У параметрі *<Формат>* вказується значення, яке має наступний синтаксис:

```
[[<Заповнювач>]<Вирівнювання>][<Знак>][#][0][<Ширина>][,]
[.<Точність>][<Перетворення>]
```

• Параметр *<Ширина>* задає мінімальну ширину поля. Якщо рядок не поміщається в зазначену ширину, то значення ігнорується, і рядок виводиться повністю:

```
>>> "'{0:10}' '(1:3)'".format(3, "string")
"'         3' '(1:3)'"
```

Ширину поля можна передати в якості параметра в методі *format()*. В цьому випадку замість числа вказується індекс параметра всередині фігурних дужок:

```
>>> "'{0:{1}}'".format(3, 10) # 10 — це ширина поля
"'         3'"
```

• Параметр *<Вирівнювання>* керує вирівнюванням значення всередині поля. Підтримуються наступні значення:

° < – по лівому краю;

° > – по правому краю (поведінка за замовчанням);

° ^ – по центру поля. Приклад:

```
>>> "{0:<10}' '{1:>10}' '{2:^10}'".format(3, 3, 3)
"3         ' '         3' '    3     '"
```

° = – знак числа вирівнюється по лівому краю, а число по правому краю:

```
>>> "'{0:=10}' '{1:=10}'".format(-3, 3)
"'-        3' '         3'"
```

Як видно з наведеного прикладу, простір між знаком і числом за замовчуванням заповнюється пробілами, а знак позитивного числа не вказується.

Щоб замість пробілів простір заповнювався нулями, необхідно вказати нуль перед шириною поля:

```
>>> "'{0:=010}' '{1:=010}'".format(-3, 3)
"'-000000003' '0000000003'"
```

• Параметр <*Заповнювач*> задає символ, яким буде заповнюватися вільний простір в поле (за замовчуванням - пробіл). Такого ж ефекту можна досягти, вказавши нуль в параметрі <*Заповнювач*>:

```
>>> "'{0:0=10}' '{1:0=10}'".format(-3, 3)
"'-000000003' '0000000003'"
>>> "'{0:*<10}' '{1:+>10}' '{2:.^10}".format(3, 3, 3)
"'3*********' '+++++++++3' '....3.....'
```

• Параметр <*Знак*> керує виводом знаку числа. Допустимі значення:

° + – задає обов'язковий вивід знака як для негативних, так і для позитивних чисел;

° - – вивід знака тільки для негативних чисел (значення за замовчуванням);

° *пробіл* – вставляє пропуск перед позитивним числом. Перед негативним числом буде стояти мінус. Приклад:

```
>>> "'{0:+}' '{1:+}' '{0:-}' '{1:-}'".format(3, -3)
"'+3' '-3' '3' '-3'"
>>> "'{0: }' '{1: }'".format(3, -3) # Пробіл
"' 3' '-3'"
```

• Для цілих чисел в параметрі <*Перетворення*> можуть бути вказані такі опції:

° *b* – перетворення в двійкову систему числення:

```
>>> "'{0:b}' '{0:#b}'".format(3)
"'11' '0b11'"
```

° *c* – перетворення числа у відповідний символ:

```
>>> "'{0:c}'".format(100)
"'d'"
```

° *x* або *X* – перетворення в шістнадцяткову систему числення в нижньому регістрі або верхньому регістрі:

```
>>> "'{0:x}' '{0:#x}' '{0:X}' '{0:#X}'".format(255)
"'ff' '0xff' 'FF' '0XFF'"
```

• Для дійсних чисел в параметрі <*Перетворення*> можуть бути вказані такі опції:

° *f* та *F* – перетворення в десяткову систему числення:

```
>>> "'{0:f}' '{1:f}' '{2:f}'".format(30, 18.6578145, -2.5)
"'30.000000' '18.657815' '-2.500000'"
```

За замовчуванням число, що виводиться має шість знаків після коми. задати іншу кількість знаків після коми ми можемо в параметрі <*Точність*>:

```
>>> "'{0:.7f}' '{1:.2f}'".format(18.6578145, -2.5)
```

```
"'18.6578145' '-2.50'"
```

° *e* або *E* – вивід в експоненційній формі (буква *e* в нижньому регістрі, буква *E* в верхньому регістрі):

```
>>> "'{0:e}' '{1:e}' '{0:E}' '{1:E}'".format(3000, 18657.81452)
"'3.000000e+03' '1.865781e+04' '3.000000E+03' '1.865781E+04'"
```

Тут за замовчанням кількість знаків після коми також дорівнює шести, але ми можемо вказати іншу величину цього параметру:

```
>>> "'{0:.2e}' '{1:.2e}' '{0:.2E}' '{1:.2E}'".format(3000, 18657.81452)
"'3.00e+03' '1.87e+04' '3.00E+03' '1.87E+04'"
```

° *g* або *G* – еквівалентно *f* або *e* (вибирається більш короткий запис числа):

```
>>> "'{0:g}' '{1:g}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578e-05'"
```

° *%* – домножує число на 100 і додає символ відсотка в кінець. Значення відображається відповідно до опції *f*:

```
>>> "'{0:%}' '{1:.4%}'".format(0.086578, 0.000086578)
"'8.657800%' '0.0087%'"
```

Швидкий тест на форматування рядків з допомогою методу *format()*. Що виведеться при виконанні таких рядків:

```
>>> print("{1:{0}}".format(3, 4))
>>> print("{0:$>5}".format(3))
>>> print("{a:{b}}".format(a = 1, b = 5))
>>> print("{a:{b}}:{0:$>5}".format(3, 4, a = 1, b = 5, c = 10))
```

Відповідь:

```
  4
$$$$3
    1
    1:$$$$3
```

1.5.3. Форматування з допомогою форматованих рядкових літералів

У Python 3.6+ з'явилася вельми зручна альтернатива методу *format()* – *форматовані рядкові літерали (Formatted String Literals)*.

Форматований рядок обов'язково повинен починатися буквою *f* або *F*. У потрібних місцях такого рядка записуються команди на вставку в ці місця значень, що зберігаються в змінних, – точно так само, як і в рядках спеціального формату, описаних раніше. Такі команди мають наступний синтаксис:

```
{[<Змінна>][!<Функція>][:<Формат>]}
```

Параметр <*Змінна*> задає ім'я змінної, з якої буде видобуто значення, що вставляється в рядок. Замість імені змінної можна записати вираз, що обчислює значення, яке потрібно вивести. Параметри <*Функція*> і <*Формат*> мають те ж призначення і записуються так само, як і у випадку методу *format()*:

```
>>> name = "Андрій"
>>> f'Привіт, {name}'
'Привіт, Андрій'
>>> value = 42
>>> message = f"The answer is {value}"
>>> print(message)
The answer is 42
>>> a = 10; b = 12.3; s = "string"
>>> f"{a} - {b} - {s}" # Простий вивід чисел і рядків
```

```
'10 - 12.3 - string'
>>> f"{a} - {b:5.2f} - {s}" # Вивід з форматуванням
'10 - 12.30 - string'
>>> d = 3
>>> f"{a} - {b:5.{d}f} - {s}" # В командах можна використовувати
                              # значення з змінних
'10 - 12.300 - string'
```

1.6. Функції і методи для роботи з рядками

1.6.1. Основні функції

Розглянемо основні функції для роботи з рядками:

♦ *str([<Об'єкт>])* – перетворює будь-який об'єкт в рядок. Якщо параметр не вказано, повертається порожній рядок. Використовується функцією *print()* для виводу об'єктів:

```
>>> str(), str([1/2]), str((3, 4)), str({"x":1})
('', '[0.5]', '(3, 4)', "{'x': 1}")
```

♦ *ascii(<Об'єкт>)* – повертає рядкове представлення об'єкту. У рядку можуть бути символи тільки з кодування ASCII:

```
>>> ascii("рядок string")
"'\\u0440\\u044f\\u0434\\u043e\\u043a string'"
```

♦ *len(<рядок>)* - повертає довжину рядка - кількість символів в ньому:

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

1.6.2. Основні методи

Наведемо перелік основних методів для роботи з рядками:

♦ *strip([<символи>])* – видаляє зазначені в параметрі символи на початку і в кінці рядка. Якщо параметр не заданий, видаляються пробільні символи: пробіл, символ переводу рядка (\n), символ повернення каретки (\r), символи горизонтальної (\t) й вертикальної (\v) табуляції:

```
>>> s1, s2 = "str\n\r\v\t", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.strip(), s2.strip("tsr"))
"'str' — 'ok'"
```

♦ *lstrip ([<символи>])* – видаляє пробільні або зазначені символи на початку рядка:

```
>>> s1, s2 = "   str    ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.lstrip(), s2.lstrip("tsr"))
"'str    ' — 'okstrstrstr'"
```

♦ *rstrip([<символи>])* – видаляє пробільні або зазначені символи в кінці рядка:

```
>>> s1, s2 = "   str    ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.rstrip(), s2.rstrip("tsr"))
"'   str' — 'strstrstrok'"
```

♦ *split([<Розділювач>[,<ліміт>]])* – розділяє рядок на підрядки за вказаним розділювачем і додає ці підрядка в список, який повертається в якості результату. Якщо перший параметр не зазначено або має значення None, то як роздільник використовується символ пробілу. У другому параметрі можна задати кількість

підрядків в результуючому списку - якщо він не вказаний або дорівнює -1, в список потраплять всі підрядки. Якщо подрядків більше зазначеної кількості, то список буде містити ще один елемент - з залишком рядка:

```
>>> s = "wordl word2 word3"
>>> s.split(), s.split(None, 1)
(['wordl', 'word2', 'word3'], ['wordl', 'word2 word3'])
>> s = "wordl\nword2\nword3"
>>> s.split("\n")
['wordl', 'word2', 'word3']
```

Якщо в рядку містяться кілька пробілів поспіль, і роздільник не вказано, то порожні елементи не будуть додані в список:

```
>>> s = "wordl     word2   word3    "
>>> s.split()
['wordl', 'word2', 'word3']
```

Якщо роздільник не знайдений в рядку, то список буде складатися з одного елемента, який представляє вихідний рядок.

♦ *rsplit([<Роздільник>[,<лімит>]])* – аналогічний методу *split()*, але пошук символу-роздільника проводиться не зліва направо, а навпаки - справа наліво:

```
>>> s = "wordl word2 word3"
>>> s.rsplit(), s.rsplit(None, 1)
(['wordl', 'word2', 'word3'], ['wordl word2', 'word3'])
```

♦ *splitlines([False])* – розділяє рядок на підрядки по символу переводу рядка (\n) і додає їх до списку. Символи нового рядка включаються в результат, тільки якщо необов'язковий параметр має значення *True*. Якщо роздільник не знайдений в рядку, список буде містити тільки один елемент:

```
>>> "wordl\nword2\nword3".splitlines()
['wordl', 'word2', 'word3']
>>> "wordl\nword2\nword3".splitlines(True)
['wordl\n', 'word2\n', 'word3']
>>> "wordl\nword2\nword3".splitlines(False)
['wordl', 'word2', 'word3']
>>> "wordl word2 word3".splitlines()
['wordl word2 word3']
```

♦ *partition(<Роздільник>)* – знаходить перше входження символу-роздільника в рядку і повертає кортеж з трьох елементів: перший елемент буде містити фрагмент, розташований перед роздільником, другий елемент сам роздільник, а третій елемент - фрагмент, розташований після роздільник. Пошук проводиться зліва направо. Якщо символ-роздільник не знайдений, то перший елемент кортежу буде містити весь рядок, а інші елементи залишаться порожніми:

```
>>> "wordl word2 word3".partition(" ")
('wordl', ' ', 'word2 word3')
```

♦ *rpartition(<Роздільник>)* – метод аналогічний методу *partition()*, але пошук символу-роздільник проводиться не зліва направо, а навпаки – справа наліво. якщо символ-роздільник не знайдений, то перші два елементи кортежу виявляться порожніми, а третій елемент буде містити всю рядок:

```
>>> "wordl word2 word3".rpartition(" ")
('wordl word2', ' ', 'word3')
```

♦ *join()* - перетворює послідовність в рядок. Елементи додаються через вказаний роздільник. Формат методу:

```
<Рядок>=<Роздільник>.join(<Послідовність>)
```

Як приклад перетворимо список і кортеж в рядок:

```
>>> " ".join(["join", "puts", "spaces", "between", "elements"])
'join puts spaces between elements'
>>> "::".join(["Separated", "with", "colons"])
'Separated::with::colons'
>>> " => ".join(("wordl", "word2", "word3"))
'wordl => word2 => word3'
```

1.6.3. Зміна регістру символів

Для зміни регістру символів призначені наступні методи:

♦ *upper()* – замінює всі символи рядка відповідними великими буквами:

```
>>> print("рядок".upper())
РЯДОК
```

♦ *lower()* – замінює всі символи рядка відповідними малими літерами:

```
>>> print("РЯДОК".lower())
рядок
```

♦ *swapcase()* – замінює усі малі символи відповідними великими буквами, а все великі символи – малими:

```
>>> print("рядок РЯДОК".swapcase())
РЯДОК рядок
```

♦ *capitalize()* – робить першу букву рядка великою:

```
>>> print("рядок рядок".capitalize())
Рядок рядок
```

♦ *title()* – робить першу букву кожного слова великою:

```
>>> s = "перша буква кожного слова стане великою"
>>> print(s.title())
Перша Буква Кожного Слова Стане Великою
```

Для роботи з окремими символами призначені наступні функції:

♦ *chr(<Код символу>)* – повертає символ за вказаним кодом:

```
>>> print(chr(1055))
П
```

♦ *ord(<Символ>)* – повертає код зазначеного символу:

```
>>> print(ord("П"))
1055
```

1.6.4. Пошук і заміна в рядку

Для пошуку і заміни в рядку використовуються такі методи:

♦ *find()* – шукає підрядок в рядку. Повертає номер позиції, з якої починається входження підрядка в рядок. Якщо підрядок в рядок не входить, то повертається значення -1. Метод залежить від регістру символів. Формат методу:

```
<Рядок>.find(<Підрядок>[,<Початок>[,<Кінець>]])
```

Якщо початкова позиція не вказана, то пошук буде здійснюватися з початку рядка. Якщо параметри *<Початок>* та *<Кінець>* вказані, то проводиться операція добування зрізу:

```
<Рядок>[<Початок>:<Кінець>]
```

і пошук підрядка буде виконуватися в цьому фрагменті:

```
>>> s = "приклад приклад Приклад"
>>> s.find("при"), s.find("При"), s.find ("тест")
```

```
(0, 16, -1)
>>> s.find("при", 12), s.find("при", 0, 6), s.find("при", 7, 12)
(-1, 0, 8)
```

♦ *rfind()* – шукає підрядок в рядку. Повертає позицію останнього входження підрядка в рядок. Якщо підрядок в рядок не входить, повертається значення -1. Метод залежить від регістру символів. Формат методу:

```
<Рядок>.rfind(<Підрядок>[,<Початок>[,<Кінець>]])
```

Якщо початкова позиція не вказана, то пошук буде проводитися з початку рядка. Якщо параметри *<Початок>* та *<Кінець>* вказані, то проводиться операція добування зрізу:

```
<Рядок>[<Початок>:<Кінець>]
```

і пошук підрядка буде виконуватися в цьому фрагменті:

```
>>> s = "приклад приклад Приклад Приклад"
>>> s.rfind("при"), s.rfind("При"), s.rfind("тест")
(8, 24, -1)
```

♦ *count()* – повертає число входжень підрядка в рядок. Якщо підрядок в рядок не входить, повертається значення 0. Метод залежить від регістру символів. Формат методу:

```
<Рядок>.count(<Підрядок>[,<Початок>[,<Кінець>]])
```

Приклад:

```
s = "приклад приклад Приклад Приклад"
>>> s.count("при"), s.count("при", 6), s.count("При")
(2, 1, 2)
```

Якщо потрібно перевірити початок чи кінець рядка на присутність певних текстових шаблонів, таких як розширення файлів, схеми URL і т. д. використовуйте методи *str.startswith()* або *str.endswith()*. Ці методи повертають результат *True* або *False* залежно від того, чи починається (або закінчується) рядок, для якого вони викликаються, одним з рядків, переданих в параметрах:

♦ *startswith()* – перевіряє, чи починається рядок з вказаного підрядка. Якщо починається, повертається значення *True*, в іншому випадку – *False*. Метод залежить від регістру символів. Формат методу:

```
<Рядок>.startswith(<Підрядок>[,<Початок>[,<Кінець>]])
```

Якщо початкова позиція не вказана, порівняння буде проводитися з початку рядка. Якщо параметри *<Початок>* та *<Кінець>* вказані, то проводиться операція добування зрізу:

```
<Рядок>[<Початок>:<Кінець>]
```

і пошук підрядка буде виконуватися з початком фрагменту:

```
>>> s = "приклад приклад Приклад Приклад"
>>> s.startswith("при"), s.startswith("При")
(True, False)
>>> url = "http://www.python.org"
>>> url.startswith("http:")
True
```

♦ *endswith()* – перевіряє, чи закінчується рядок зазначеним підрядком. Якщо закінчується, то повертається значення *True*, в іншому випадку – *False*. Метод залежить від регістру символів. Формат методу:

```
<Рядок>. endswith(<Підрядок>[,<Початок>[,<Кінець>]])
```

Якщо початкова позиція не вказана, порівняння буде проводитися з кінцем рядку. Якщо параметри <*Початок*> та <*Кінець*> вказані, то проводиться операція добування зрізу:

```
<Рядок>[<Початок>:<Кінець>]
```

і пошук підрядка буде виконуватися з кінцем фрагменту:

```
>>> s = "підрядок ПІДРЯДОК"
>>> s.endswith("док"), s.endswith("ДОК")
(False, True)
>>> s.endswith("док", 0, 8)
True
>>> filename = "spam.txt"
>>> filename.endswith(".txt")
True
```

Методи *startswith()* та *endswith()* дають змогу шукати більше одного рядка одночасно і повертають *True*, якщо знайдено хоча б один:

```
>>> filename = "spam.txt"
>>> filename.endswith((".txt", ".doc", ".pdf"))
True
```

Методи *startswith()* та *endswith()* надають досить зручний спосіб перевірки префіксів і закінчень. Такі ж операції можна зробити і з допомогою зрізів, але значно менш елегантно:

```
>>> url[:5] == "http:"
True
>>> filename[-4:] == ".txt"
True
```

♦ *replace()* – робить заміну всіх входжень заданого підрядка в рядку на інший підрядок і повертає результат у вигляді нового рядка. Метод залежить від регістру символів. Формат методу:

```
<Рядок>.replace(<Підрядок для заміни>,<Новий підрядок>[,
                <Максимальна кількість замін>])
```

Якщо кількість замін не вказано, буде виконано заміну всіх знайдених підрядків:

```
>>> s = "Привіт, Петро"
>>> print(s.replace("Петро", "Олег"))
Привіт, Олег
>>> x = "Mississippi"
>>> x.replace("ss", "+++")
'Mi+++i+++ippi'
```

1.6.5. Перевірка типу вмісту рядка

Для перевірки типу вмісту призначені наступні методи:

♦ *isalnum()* – повертає *True*, якщо рядок містить лише літери і (або) цифри, в іншому випадку – False. Якщо рядок порожній, повертається значення *False*:

```
>>> "0123".isalnum(), "abc".isalnum(), "abcl23".isalnum()
 (True, True, True)
>>> "".isalnum(), "123 abc".isalnum(), "abc, 123.".isalnum()
(False, False, False)
```

♦ *isalpha()* – повертає *True*, якщо рядок містить лише літери, в іншому випадку – *False*. Якщо рядок порожній, повертається значення *False*:

```
>>> "string".isalpha (), "рядок".isalpha(), "".isalpha()
(True, True, False)
>>> "123abc".isalpha(), "str str".isalpha(), "st,st".isalpha()
(False, False, False)
```

♦ *isdigit()* – повертає *True*, якщо рядок містить лише цифри, в іншому випадку – *False*:

```
>>> "0123".isdigit(), "123abc".isdigit(), "abcl23".isdigit()
(True, False, False)
```

♦ *isupper()* – повертає *True*, якщо рядок містить літери тільки верхнього регістру, в іншому випадку – *False*:

```
>>> "ST".isupper(), "РД, 1".isupper(), "".isupper(), "12".isupper()
(True, True, False, False)
```

♦ *islower()* – повертає *True*, якщо рядок містить літери тільки нижнього регістру, в іншому випадку - *False*:

```
>>> "st".islower(), "ст, 1".islower(), " ".islower(), "12".islower()
(True, True, False, False)
```

♦ *istitle()* – повертає *True*, якщо всі слова в рядку починаються з великої літери, в іншому випадку – *False*. Якщо рядок порожній, також повертається *False*:

```
>>> "Str Str, 123".istitle(), "Str str".istitle(), "123".istitle()
(True, False, False)
```

♦ *isprintable()* – повертає *True*, якщо рядок містить тільки друковані символи, в іншому випадку – *False*. Пробіл вважається друкованим символом:

```
>>> "123".isprintable(), "Yes No".isprintable(), "\n".isprintable()
(True, True, False)
```

♦ *isspace()* – повертає *True*, якщо рядок містить лише пробільні символи, в іншому випадку – *False*:

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
(False, True, False)
```

♦ *isidentifier()* – повертає *True*, якщо рядок представляє собою допустимі з точки зору Python ім'я змінної, функції або класу, в іншому випадку – *False*:

```
>>> "s".isidentifier(), "1func".isidentifier()
(True, False)
```

Слід мати на увазі, що метод *isidentifier()* лише перевіряє, чи задовольняє задане ім'я правилами мови. Він не перевіряє, чи збігається це ім'я з ключовим словом Python.

Модуль *string* визначає ряд корисних констант при роботі з рядками. Константа *string.whitespace* – рядок, яка складається з усіх символів, які Python відносить до категорії пробілів у вашій системі.

```
>>> import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

Константа *string.digits* містить рядок зі всіма цифрами '0123456789':

```
>>> string.digits
'0123456789'
```

Константа *string.hexdigits* включає всі символи *string.digits*, а також *'abcdefABCDEF'* – додаткові символи, використовувані в шістнадцятирічних числах:

```
>>> string.hexdigits
```

```
'0123456789abcdefABCDEF'
```

Константа *string.ascii_lowercase* містить всі алфавітні символи ASCII нижнього регістру:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Константа *string.ascii_uppercase* містить всі алфавітні символи ASCII верхнього регістру:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Константа *string.ascii_letters* містить всі символи *string.ascii_lowercase* і *string.ascii_uppercase*:

```
>>> string. ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

2. ЗАВДАННЯ

2.1. Домашня підготовка до роботи

1. Вивчити теоретичний матеріал.

2.2. Виконати в лабораторії

1. Написати програму, яка для заданого тексту використовуючи арифметичні і рядкові операції та операції форматування тексту виводить відформатований текст. Форматування здійснити трьома способами: з допомогою оператора %, методу *format()* і форматованих рядкових літералів.

Текст для форматування: «Сьогодні в "a" буде проходити позачергове засідання комітету з проблем "b", організоване "c". Було подано "d" заявок на загальну суму "e" тис. гривень. Середня вартість проекту склала "g" гривень.»

Наприклад: "a" – 12.30, "b" - «Інформатизація освіти», "c" – МОНУ, "d" – 213, "e" – 123000, "g" = e/d – з точністю до 2-х знаків після коми.

Параметри вводяться з клавіатури в діалоговому режимі.

2. Написати програму валідації введеного паролю. Програма не повинна використовувати умовні оператори (*if* та інші), цикли, регулярні вирази, списки, множини, словники та інші структури даних, функції, класи чи сторонні бібліотеки окрім *colorama*. Користувач повинен ввести пароль, програма має перевірити наявність у ньому різних типів символів (хоча б одного кожного заданого типу) згідно варіанту у табл. 1 і вивести інформацію про результати перевірки у форматі як показано на рис. 1.

Номер варіанту відповідає номеру в списку групи.

```
Введіть пароль довжиною не менше 6 символів.
Вимоги до паролю:
1. Великі літери
2. Маленькі літери
3. Цифри
4. Спеціальні символи !@#$_-%^&*
> 01234
Довжина не менше 6 символів - FAIL!
Великі літери - FAIL!
Маленькі літери - FAIL!
Цифри - OK!
Спеціальні символи - FAIL!

Пароль не валідний!

Введіть пароль довжиною не менше 6 символів.
Вимоги до паролю:
1. Великі літери
2. Маленькі літери
3. Цифри
4. Спеціальні символи !@#$_-%^&*
> dfghAA45&*=
Довжина не менше 6 символів - OK!
Великі літери - OK!
Маленькі літери - OK!
Цифри - OK!
Спеціальні символи - OK!

Пароль валідний!

Введіть пароль довжиною не менше 6 символів.
Вимоги до паролю:
1. Великі літери
2. Маленькі літери
3. Цифри
4. Спеціальні символи !@#$_-%^&*
> мій_пароль!!!))
Довжина не менше 6 символів - OK!
Великі літери - FAIL!
Маленькі літери - OK!
Цифри - FAIL!
Спеціальні символи - OK!

Пароль не валідний!
```

Рис. 1. Оформлення вводу-виводу інформації під час валідації пароля

3. Написати програму генерації паролю. Програма не повинна використовувати умовні оператори (*if* та інші), цикли, регулярні вирази, функції, класи чи сторонні бібліотеки окрім *random*. Користувач повинен ввести кількість різних типів символів у паролі згідно варіанту у табл. 2 і вивести згенерований пароль у форматі як показано на рис. 2.

```
Петренко Олег Степанович, КБ-101, 2024. Варіант 16

Введіть кількість великих латинських літер в паролі: 3
Введіть кількість малих латинських літер в паролі: 3
Введіть кількість цифр в паролі: 2
Введіть кількість спеціальних символів !@#$_-%^&* в паролі: 1
Password: b1JX&w9Cs
```

Рис. 2. Оформлення вводу-виводу інформації під час генерації пароля

4. Задані імена файлів у каталозі у вигляді рядка:

```
dir = (
    "_file1.doc\n"
    "file2.pdf\n"
    "file222_.docx\n"
    "cmd.exe\n"
    "sys.dll\n"
    "FiLe7_5.txt\n"
    "foto1.jpg\n"
    "song1.mp3\n"
    "!!!song2.mp3\n"
    "video.avi\n"
    "file9.txt\n"
    "file_3_document.docx\n"
    "my_document!!!.ppt\n"
    "main.c\n"
    "lab3.py\n"
    "lookup.xml\n"
    "pic1.png\n"
    "pic2.bmp\n"
)
```

Написати програму яка б:

1. Підраховувала і виводила кількість файлів у каталозі.

2. Підраховувала і виводила кількість файлів з заданими розширеннями згідно стовпця **A** табл. 3.

3. Перейменовувала розширення у всіх файлів згідно стовпця **B** табл. 3.

4. Приводила імена всіх файлів до заданого регістру згідно стовпця **C** табл. 3.

5. Видаляла всі файли з заданим розширенням згідно стовпця **D** табл. 3.

Оформлення виводу інформації має виглядати таким чином:

```
Початковий вміст каталогу
_file1.doc
file2.pdf
…
pic1.png
pic2.bmp

У каталозі є 18 файлів
Файлів з розширенням docx: 2
Файлів з розширенням c: 1
```

```
Каталог після заміни розширення .doc на .docx
_file1.docx
file2.pdf
file222_.docx
…
pic1.png
pic2.bmp

Каталог після приведення імен файлів до верхнього регістру
_FILE1.DOCX
FILE2.PDF
…
PIC1.PNG
PIC2.BMP

Каталог після видалення файлу PIC1.PNG
_FILE1.DOCX
FILE2.PDF
…
LOOKUP.XML
PIC2.BMP
```

3. ЗМІСТ ЗВІТУ

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

У якості наборів символів можуть виступати великі (upp_char) і малі (low_char) літери, цифри (num_char) і спеціальні символи (spc_char = "!@#$%^&*_-").

Табл. 1

Варіанти завдань

Варіант	Довжина, символів	Набір символів
1.	12	low_char + spc_char + upp_char
2.	15	low_char + num_char + upp_char
3.	10	low_char + num_char + spc_char + upp_char
4.	9	spc_char + upp_char
5.	12	low_char + num_char + spc_char + upp_char
6.	10	low_char + num_char + spc_char
7.	14	low_char + num_char + spc_char + upp_char
8.	9	low_char + upp_char
9.	8	num_char + spc_char
10.	9	low_char + spc_char

11.	8	num_char + spc_char + upp_char
12.	11	low_char + spc_char + upp_char
13.	14	low_char + num_char + spc_char + upp_char
14.	10	low_char + spc_char + upp_char
15.	11	low_char + num_char + spc_char + upp_char
16.	13	low_char + num_char + spc_char + upp_char
17.	13	low_char + num_char + spc_char + upp_char
18.	11	low_char + upp_char
19.	12	low_char + num_char + spc_char + upp_char
20.	11	low_char + spc_char + upp_char
21.	6	num_char + spc_char
22.	14	low_char + num_char + spc_char + upp_char
23.	9	low_char + num_char + spc_char + upp_char
24.	10	spc_char + upp_char
25.	9	low_char + num_char + upp_char
26.	7	num_char + upp_char
27.	10	low_char + num_char + spc_char + upp_char
28.	15	low_char + num_char + spc_char + upp_char
29.	8	num_char + spc_char + upp_char
30.	13	low_char + num_char + upp_char

upp_char = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
low_char = "abcdefghijklmnopqrstuvwxyz"
num_char = "0123456789"
spc_char = "!@#$%^&*_-"

У душках в табл. 2 вказано кількість символів з даного набору. Напр. low_char(3) означає, що пароль має містити 3 маленькі латинські літери.

Табл. 2

Варіанти завдань

Варіант	**Довжина паролю, символів**	**Набір символів**
1.	12	low_char(5) + spc_char(3) + upp_char(4)
2.	15	low_char(7) + num_char(4) + upp_char(4)
3.	7	low_char(3) + num_char(2) + spc_char(1) + upp_char(1)
4.	7	spc_char(2) + upp_char(5)
5.	8	low_char(3) + num_char(3) + spc_char(1) + upp_char(1)
6.	7	low_char(2) + num_char(3) + spc_char(2)
7.	14	low_char(6) + num_char(3) + spc_char(2) + upp_char(3)
8.	9	low_char(5) + upp_char(4)
9.	12	num_char(7) + spc_char(5)
10.	13	low_char(9) + spc_char(4)
11.	8	num_char(3) + spc_char(2) + upp_char(3)
12.	10	low_char(4) + spc_char(2) + upp_char(4)
13.	14	low_char(5) + num_char(4) + spc_char(2) + upp_char(3)
14.	9	low_char(3) + spc_char(1) + upp_char(5)
15.	8	low_char(2) + num_char(1) + spc_char(1) + upp_char(4)

16.	13	low_char(4) + num_char(3) + spc_char(2) + upp_char(4)
17.	13	low_char(3) + num_char(5) + spc_char(3) + upp_char(2)
18.	11	low_char(7) + upp_char(4)
19.	11	low_char(2) + num_char(1) + spc_char(2) + upp_char(6)
20.	11	low_char (2)+ spc_char(2) + upp_char(7)
21.	6	num_char(4) + spc_char(2)
22.	12	low_char(6) + num_char(1) + spc_char(2) + upp_char(3)
23.	9	low_char(3) + num_char(3) + spc_char(2) + upp_char(1)
24.	13	spc_char(4) + upp_char(9)
25.	9	low_char(4) + num_char(3) + upp_char(2)
26.	7	num_char(5) + upp_char(2)
27.	8	low_char(4) + num_char(2) + spc_char(1) + upp_char(1)
28.	15	low_char(5) + num_char(3) + spc_char(2) + upp_char(5)
29.	7	num_char(3) + spc_char(2) + upp_char(2)
30.	13	low_char(8) + num_char(3) + upp_char(2)

Стовбець **A** – для підрахунку файлів з заданими розширеннями.
Стовбець **B** – для перейменування файлів з одного розширення на інше.
Стовбець **C** – задає найменування файлу у верхньому регістрі (U), нижньому регістрі (L) або в нижньому регістрі з першою великою літерою (T).
Стовбець **D** – задає розширення файлів, які потрібно видалити.

Табл. 3

Варіант	**A**	**B**	**C**	**D**
1.	c + py	avi --> exe	U	exe
2.	docx + png	exe --> ppt	L	ppt
3.	bmp + ppt	doc --> png	T	png
4.	jpg + txt	ppt --> pdf	U	pdf
5.	doc + py	txt --> doc	L	doc
6.	c + exe	pdf --> avi	T	avi
7.	bmp + docx	doc --> avi	U	avi
8.	bmp + xml	jpg --> png	L	png
9.	mp3 + pdf	ppt --> exe	T	exe
10.	ppt + txt	bmp --> jpg	U	jpg
11.	ppt + py	ppt --> docx	L	docx
12.	jpg + py	doc --> mp3	T	mp3
13.	dll + docx	dll --> jpg	U	jpg
14.	jpg + py	c --> pdf	L	pdf
15.	c + xml	ppt --> mp3	T	mp3
16.	docx + exe	c --> mp3	U	mp3
17.	doc + xml	txt --> png	L	png

18.	dll + docx	bmp --> jpg	T	jpg
19.	c + txt	py --> xml	U	xml
20.	dll + mp3	doc --> py	L	py
21.	c + dll	jpg --> png	T	png
22.	exe + jpg	txt --> mp3	U	mp3
23.	avi + mp3	ppt --> xml	L	xml
24.	avi + py	mp3 --> bmp	T	bmp
25.	bmp + docx	avi --> txt	U	txt
26.	bmp + ppt	dll --> exe	L	exe
27.	mp3 + png	avi --> xml	T	xml
28.	bmp + pdf	dll --> txt	U	txt
29.	bmp + exe	ppt --> dll	L	dll
30.	exe + jpg	png --> avi	T	avi

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Чи можна змінювати рядки ?
2. Які є методи пошуку і заміни для рядків ?
3. Назвіть основні методи розбиття рядків ?
4. Як створюються рядки ?
5. Які є способи форматування рядків та вкажіть їх основні можливості ?

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.
6. https://www.hivesystems.com/blog/are-your-passwords-in-the-green

НАВЧАЛЬНЕ ВИДАННЯ

Робота з рядками

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 3
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

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