



# ОСНОВИ РОБОТИ З ФАЙЛОВОЮ СИСТЕМОЮ ЗАСОБАМИ PYTHON

Тема 09

# План лекції

---

- Робота з рядками та регулярні вирази.
- Основи файлового вводу-виводу в Python.
- Серіалізація об'єктів у мові Python. Піклінг.
- Робота з іншими серіалізаційними представленнями.



# РОБОТА З РЯДКАМИ ТА РЕГУЛЯРНІ ВИРАЗИ

Питання 9.1

# Операції з рядками. Клас str

---

a = "hello"

Останній рядок автоматично компонується в єдине ціле інтерпретатором.

b = 'world'

Можна виконувати конкатенацію за допомогою оператора + ("hello " + "world").

c = "a multiple

Рядки не обов'язково повинні бути жорстко вписаними в коді.

line string"

- Вони можуть надходити від зовнішніх джерел: текстових файлів, вводу користувача, з мережі тощо.

d = ""More

Кілька перевірочних Boolean-методів допомагають визначити, чи відповідають (match) символи в рядку деякому шаблону.

multiple""

- Очевидна функціональність в isalpha(), isupper()/islower(), startswith()/endswith().
- Метод isspace() перевіряє пробільні символи, в тому числі табуляцію, перехід на новий рядок тощо.

e = ("Three " "Strings "

"Together")

- Метод istitle() повертає True, якщо перший символ кожного слова є великими літерами, а решта – малими. Жорстких обмежень англійської граматики немає. Наприклад, вірш Leigh Hunt "The Glove and the Lions" вважатиметься коректним заголовком, хоч не всі слова з великої букви.

# Методи `isdigit()`, `isdecimal()`, `isnumeric()` мають багато нюансів

---

- Застосуємо ці методи:

```
>>> s = "hello world"
>>> s.count('l')
3
>>> s.find('l')
2
>>> s.rindex('m')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```
- Багато Unicode-символів вважаються числами.
  - Крапка в дробових числах не вважається символом десяткового числа, тому `'45.2'.isdecimal()` повертає `False`.
  - Справжній десятковий символ представлений в Unicode значенням `об6о – 45\уоб6о2`.
  - Ці методи не перевіряють, чи є число в рядку коректним: `"127.о.о.1"` поверне `True` для їх усіх.
  - Використовувати `'\уоб6о'` замість крапки теж не вихід, оскільки парсинг числа за допомогою `float()` або `int()` конвертує десятковий символ у нуль:

```
>>> float('45\уоб6о2')
4502.0
```
- Метод `count()` визначає, скільки разів заданий підрядок зустрічається в рядку, а `find()`, `index()`, `rfind()` та `rindex()` повертають позицію підрядка в початковому рядку.
  - Два `'r'`-методи починають пошук з кінця рядка.
  - Методи `find` повертають `-1`, якщо підрядок не знайдено, а `index` викликає `ValueError` в такій ситуації.

# Більшість з решти методів для роботи з рядками повертають трансформований рядок

---

- Методи `upper()`, `lower()`, `capitalize()`, `title()` створюють нові рядки з алфавітними символами в заданому форматі.
  - Метод `translate` може використовувати словник, щоб відобразити довільно введені символи в заданий формат виводу.
- Ці методи повертають новий екземпляр `str`, тому для роботи з ним потрібно вводити змінну.
  - Наприклад, `new_value = value.capitalize()`.
- Метод `split()` приймає підрядок та розбиває рядок на список рядків, які розділялися заданим підрядком.
  - Можна передати число в якості другого параметру – обмеження на кількість елементів списку.
  - Метод `rsplit` поводитьсь так же, проте починає розбиття рядка з кінця, якщо введено обмеження.
- Методи `partition()` та `rpartition()` розбивають рядок лише в першому та останньому знаходженні підрядка.
  - Повертають кортеж з трьох значень: символи до підрядка, сам підрядок та символи після нього.
- Оберненим до `split()` є метод `join()`.
  - Приймає список рядків, а повертає комбінацію з елементів списку.
  - Метод `replace` отримує 2 аргументи та повертає рядок, у якому кожен екземпляр першого аргументу було замінено на другий.

## Деякі з цих методів у дії

---

- `>>> s = "hello world, how are you"`
- `>>> s2 = s.split(' ')`
- `>>> s2`
- `['hello', 'world,', 'how', 'are', 'you']`
- `>>> '#'.join(s2)`
- `'hello#world,#how#are#you'`
- `>>> s.replace(' ', '**')`
- `'hello**world,**how**are**you'`
- `>>> s.partition(' ')`
- `('hello', ' ', 'world, how are you')`

# Форматування рядків

---

- Використовується метод `format()`.
  - Повертає новий рядок, в якому символи вхідного тексту замінено на значення аргументів and keyword arguments passed into the function.
  - Метод не потребує фіксованого набору аргументів - всередині використовується синтаксис `*args` та `**kwargs`.
- Спеціальні символи на заміну – це `{}` та `%`.
  - We can insert pairs of these in a string and they will be replaced, in order, by any positional arguments passed to the `str.format` method:
  - `template = "Hello {}, you are currently {}".format('Dusty', 'writing')`
  - If we run these statements, it replaces the braces with variables, in order:
  - Hello Dusty, you are currently writing.



- 
- Базовий синтаксис дуже корисний, коли потрібно повторно використати змінні з одного рядка чи в різних позиціях.
    - Спробуємо повторити ім'я:
      - `template = "Hello {0}, you are {1}. Your name is {0}."`
      - `print(template.format('Dusty', 'writing'))`
  - Якщо беруться цілочисельні індекси, їх потрібно використовувати для всіх змінних.
    - Змішувати порожні фігурні дужки з позиційними індексами не можна.
    - Такий код викине `ValueError` :
      - `template = "Hello {}, you are {}. Your name is {0}."`
      - `print(template.format('Dusty', 'writing'))`
  - Фігурні дужки корисні для роботи з рядком не тільки при форматуванні.
    - Вивести на екран саме фігурну дужку можна шляхом її дублювання.
    - Наприклад, вивід відформатованої Java-програми за допомогою Python:

```
template = ""  
public class {0} {{  
    public static void main(String[] args) {{  
        System.out.println("{1}");  
    }}  
}}""  
  
print(template.format("MyClass", "print('hello world')"));
```



```
public class MyClass {  
    public static void main(String[] args) {  
        System.out.println("print('hello world')");  
    }  
}
```

# Іменовані (Keyword) аргументи

---

```
template = """
From: <{from_email}>
To: <{to_email}>
Subject: {subject}

{message}"""
print(template.format(
    from_email = "a@example.com",
    to_email = "b@example.com",
    message = "Here's some mail for you. "
    " Hope you enjoy the message!",
    subject = "You have mail!"
))
```

- При форматуванні складних рядків пам'ятати порядок аргументів або оновити його після вставки нового аргументу може бути незручно.
  - Метод `format` дозволяє задавати назви всередині фігурних дужок.
  - Іменовані змінні потім передаються в метод як `keyword arguments`.
- Можна комбінувати індекси та `keyword`-аргументи.
  - І навіть немічені позиційні фігурні дужки з `keyword`-аргументами:
  - `print("{} {} {}".format("x", "y", label="z"))`

# Перегляди контейнерів

---

- Будь-який примітивний тип, зокрема integer або float, можна вивести на екран.
  - До змінних та індексів складених об'єктів (списків, кортежів, словників та довільних об'єктів) можна отримати доступ з format string.

```
emails = ("a@example.com", "b@example.com")
message = {
    'subject': "You Have Mail!",
    'message': "Here's some mail for you!"
}
template = """
From: <{0[0]}>
To: <{0[1]}>
Subject: {message[subject]}
{message[message]}"""
print(template.format(emails, message=message))
```

- Було передано один аргумент як позиційний параметр, а інший – як keyword argument.
  - Дві електронні адреси переглядаються (look up) by o[x], де x є 0 або 1.
  - Нуль представляє, as with other position-based arguments, перший позиційний аргумент, переданий у format (тут – кортеж електронних адрес).
- Квадратні дужки з числом всередині мають стандартне значення звернення за індексом, тому o[o] відображається в emails[o] в кортежі адрес.
  - На відміну від стандартного Python-коду, не потрібно оточувати лапками рядок у dictionary lookup.

# Можна робити кілька рівнів звернень для вкладених структур даних

---

- Використовувати не дуже рекомендується, оскільки шаблонні рядки дуже швидко стають незрозумілими.
  - Якщо маємо словник, що містить кортеж, можемо зробити таке:

```
emails = ("a@example.com", "b@example.com")
message = {
    'emails': emails,
    'subject': "You Have Mail!",
    'message': "Here's some mail for you!"
}
template = """
From: <{0[emails][0]}>
To: <{0[emails][1]}>
Subject: {0[subject]}
{0[message]}"""
print(template.format(message))
```

# Звернення до об'єктів (Object lookups)

---

- Також можна передавати довільні об'єкти в якості параметрів, а для доступу до атрибутів використовувати dot notation.
  - Знову змінимо дані електронного повідомлення:

```
1 class Email:
2     def __init__(self, from_addr, to_addr, subject, message):
3         self.from_addr = from_addr
4         self.to_addr = to_addr
5         self.subject = subject
6         self.message = message
7
8 email = Email("a@example.com", "b@example.com",
9              "У Вас нове повідомлення!", "Ось деякі повідомлення!")
10 template = """
11 From: <{0.from_addr}>
12 To: <{0.to_addr}>
13 Subject: {0.subject}
14 {0.message}"""
15 print(template.format(email))
```

```
From: <a@example.com>
To: <b@example.com>
Subject: У Вас нове повідомлення!
Ось деякі повідомлення!
```

# Making it look right

---

- Форматування валют:

```
1 subtotal = 12.32
2 tax = subtotal * 0.07
3 total = subtotal + tax
4 print("Sub: ${0} Tax: ${1} Total: ${total}".format(subtotal, tax, total=total))
```

- Результати виводу не підходять для роботи з валютами:

```
Sub: $12.32 Tax: $0.8624 Total: $13.182400000000001
```

- Слід використовувати специфікатори формату

```
print("Sub: ${0:0.2f} Tax: ${1:0.2f} "  
      "Total: ${total:0.2f}".format(  
      subtotal, tax, total=total))
```

- Технічно, ніколи не використовуйте дробові числа у грошових обчисленнях; краще конструювати об'єкти `decimal.Decimal()`.

## Додаткові можливості форматування

---

```
1 orders = [('burger', 2, 5),
2           ('fries', 3.5, 1),
3           ('cola', 1.75, 3)]
4 print("PRODUCT  QUANTITY  PRICE  SUBTOTAL")
5 for product, price, quantity in orders:
6     subtotal = price * quantity
7     print("{0:10s}{1: ^9d} ${2: <8.2f}${3: >7.2f}".format(
8         product, quantity, price, subtotal))
```

### ■ Результати виводу:

PRODUCT	QUANTITY	PRICE	SUBTOTAL
burger	5	\$2.00	\$ 10.00
fries	1	\$3.50	\$ 3.50
cola	3	\$1.75	\$ 5.25

- Специфікатори мають бути в правильному порядку, хоч усі вони опційні: спочатку fill, потім align, далі size, а за ним type.

Маємо 4 змінні для форматування:

- {0:10s} – рядок (s) на 10 символів
- {1: ^9d} – ціле число (d), займає до 9 розрядів. Невикористані розряди заповнюються нулями, тому символ ^ вирівнює число по центру.
- {2: <8.2f}, {3: >7.2f} – числа з вирівнюванням по лівій (<) або правій (>) сторонах.

# Символ "type" для різних типів може впливати на форматування виводу

---

- Були символи s, d та f для рядків, цілих та дробових чисел відповідно.
  - Більшість інших специфікаторів формату – їх альтернативні версії.
  - Наприклад, o представляє вісімковий формат, а X – шістнадцятковий.
  - Специфікатор n корисний при форматуванні integer separators in the current locale's format.
  - Для дробових чисел %type помножить їх на 100 та format a float as a percentage.
- Також можливо визначити нестандартні специфікатори для об'єктів.
  - Наприклад, при передачі об'єкту datetime у format() можна використовувати специфікатори у функції datetime.strftime()

```
import datetime
print("{0:%Y-%m-%d %I:%M%p }".format(
    datetime.datetime.now() ) )
```



# Рядки закодовано в Unicode

---

- Це колекції незмінюваних Unicode-символів.
  - Отримуючи рядок байтів з файлу чи сокета, він не буде в Unicode.
  - Байти не представляють щось конкретне; може бути рядок, пікселі зображення та ін.
  - При виводі байтового об'єкта байти перетворюються на ASCII-символи, якщо їх значення потрапляють у межі таблиці.
  - Не-ASCII байти (двійкові дані або інші символи) виводяться як шістнадцяткові коди.
  - Багато операцій вводу-виводу знають, як працювати з байтами, навіть якщо байти відносяться до текстових даних.
  - Важливо знати, як конвертувати байти та Unicode-символи.

# Конвертування байтів у текст

---

```
1 characters = b'\x63\x6c\x69\x63\x68\xe9'
2 print(characters)
3 print(characters.decode("latin-1"))
```

- Послідовність байтів (hex) 63 6c 69 63 68 e9 представляє слово cliché в кодуванні latin-1.
- Результати виводу:

```
b'clich\xe9'
clich  
```

  - Перший print виводить байти у вигляді ASCII-символів.
  - Нерозпізнаний символ залишається у шістнадцятковому форматі.
  - Символ b на початку рядка нагадує про байтове представлення, а не рядкове.
- Якби ми використовували кириличне кодування "iso8859-5«, а не latin-1, то отримали б рядок 'clіchщ'.

# Конвертування тексту в байти

---

```
1 characters = "clich  "
2 print(characters.encode("UTF-8"))
3 print(characters.encode("latin-1"))
4 print(characters.encode("CP437"))
5 print(characters.encode("ascii"))
```

## ■ Результати виводу:

```
b'clich\x3\xa9'
```

```
b'clich\xe9'
```

```
b'clich\x82'
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-23-666c189f55f5>", line 1, in <module>
    runfile('C:/Users/spuasson/Desktop/untitled6.py', wdir='C:/Users/spuasson/Desktop')
```

```
File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 710, in runfile
    execfile(filename, namespace)
```

```
File "C:\ProgramData\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)
```

```
File "C:/Users/spuasson/Desktop/untitled6.py", line 5, in <module>
    print(characters.encode("ascii"))
```

```
UnicodeEncodeError: 'ascii' codec can't encode character '\xe9' in position 5: ordinal not in range(128)
```

# Обробка виключень при зустрічі невідомого символу

---

- Метод `encode` приймає опційний рядковий аргумент, який визначає, як обробляти помилки представлення символів.
  - `strict` – стандартна стратегія обробки: викидання виключення.
  - `replace` – невідомий символ замінюється на інший (в ASCII на '?').
  - `ignore` – відкидає всі незрозумілі символи.
  - `xmlcharrefreplace` – створює xml-сутність, яка представляє Unicode-символ

Strategy	"cliché".encode("ascii", strategy)
<code>replace</code>	<code>b'clich?'</code>
<code>ignore</code>	<code>b'clich'</code>
<code>xmlcharrefreplace</code>	<code>b'clich&amp;#233;'</code>

- Можна викликати методи `str.encode` та `bytes.decode` без передачі `encoding string`.
  - Кодування буде обрано за замовчуванням для поточної платформи – залежно від ОС або регіональних налаштувань; перевірити можна за допомогою функції `sys.getdefaultencoding()`.
  - Зазвичай хороша ідея – явно задати кодування, оскільки кодування за замовчуванням може змінитись, або програма одного дня розширить свій набір джерел надходження інформації.

# Якщо кодування тексту невідоме, краще брати UTF-8

---

- UTF-8 здатне представити будь-який символ Unicode.
  - Нині це фактичний стандарт кодування документів будь-якою мовою.
  - Інші можливі кодування корисні для застарілих документів or in regions that still use different character sets by default.
- Кодування UTF-8 використовує 1 байт для представлення ASCII та інших поширених символів і до 4 байтів для складніших символів.
  - UTF-8 – особливе кодування, зворотно сумісне з ASCII;
  - Будь-який ASCII-документ закодований в UTF-8 will be identical to the original ASCII document.

# Змінювані байтові рядки

---

- Байтовий тип, як і `str`, незмінюваний (`immutable`).
  - Можна використовувати індексну та `slice`-нотації для байтових об'єктів та шукати конкретну послідовність байтів, проте можливості доповнювати чи змінювати їх немає.
  - При вводі-виводі це незручно, оскільки вхідні та вихідні байти часто буферизуються, поки не будуть надіслані.
  - Наприклад, при надходженні даних від сокету може відбутись кілька викликів `recv` до того, як отримаємо все повідомлення.
- Тут допомагають вбудовані байтові масиви.
  - Тип `bytearray` на зразок списку, проте з байтів.
  - Конструктор класу може приймати байтові об'єкти для ініціалізації.
  - Метод `extend()` використовується для дописування байтових об'єктів до існуючого масиву (наприклад, `when more data comes from a socket or other I/O channel`).

# Slice-нотація може використовуватись для bytearray, щоб змінювати його

---

- Наприклад, код конструює bytearray з байтового об'єкту та замінює 2 байти:

```
1 b = bytearray(b"abcdefgh")
2 b[4:6] = b"\x15\xa3"
3 print(b)
```

- **Вивід:** bytearray(b'abcd\x15\xa3gh')

- Обережно: при бажанні оперувати одним елементом байтового масиву очікується передача цілого числа з діапазону від 0 до 255 включно.
- Дане ціле число представляє конкретний байтовий шаблон (pattern).
- При спробі передачі символу чи байтового об'єкта викинеться виключення.

- Один байтовий символ можна конвертувати в ціле число за допомогою функції ord:

- Функція повертає цілочисельне представлення одного символу.
- **Результат виводу:** bytearray(b'abcgDf')
- Після конструювання масиву замінюємо символ за індексом 3 на byte 103.
- Повернене функцією ord() ціле число є ASCII-символом для літери 'g'.

```
1 b = bytearray(b'abcdef')
2 b[3] = ord(b'g')
3 b[4] = 68
4 print(b)
```

# Регулярні вирази (regular expressions)

---

- Парсинг рядків для пошуку текстових шаблонів складно реалізується в контексті об'єктно-орієнтованих принципів.
  - Для цього більшість мов програмування використовують регулярні вирази.
  - Хоч вони не об'єктно-орієнтовані, бібліотека Python для регулярних виразів забезпечує кілька класів та об'єктів, які використовуються для конструювання та роботи РВ.
- Регулярні вирази вирішують поширену задачу:
  - Для заданого рядка визначити, чи відповідає він заданому шаблону та, опційно, збирати підрядки, які містять релевантну інформацію.
- РВ відповідають на запитання:
  - Чи є рядок коректним URL-посиланням?
  - Які дата та час усіх попереджень у log-файлі?
  - Які з користувачів з/etc/passwd знаходяться в деякій групі?
  - Які username та document були запитані по URL, введеному відвідувачем?



# Зіставлення з шаблонами (Matching patterns)

---

```
1 import re
2
3 search_string = "hello world"
4 pattern = "hello world"
5
6 match = re.match(pattern, search_string)
7
8 if match:
9     print("regex matches")
```

- Регулярні вирази використовують спеціальні символи, щоб шукати текстові шаблони в заздалегідь невідомих рядках.
  - Після імпорту модуля `re` задаємо рядок пошуку та текстовий шаблон (`pattern`) для пошуку.
  - Якщо з'являється співпадіння, спрацьовує `print()`.
  - Функція `match()` виконує зіставлення з шаблоном з початку рядка. Тому за шаблоном "ello world" нічого не буде знайдено.

```
1 import sys
2 import re
3
4 pattern = sys.argv[1]
5 search_string = sys.argv[2]
6 match = re.match(pattern, search_string)
7 if match:
8     template = "'{}' matches pattern '{}'"
9 else:
10    template = "'{}' does not match pattern '{}'"
11 print(template.format(search_string, pattern))
```

```
$ python regex_generic.py "hello worl" "hello world"
'hello world' matches pattern 'hello worl'
```

```
$ python regex_generic.py "ello world" "hello world"
'hello world' does not match pattern 'ello world'
```

- 
- 
- За потреби керування тим, чи зустрічатимуться символи з початку чи кінця рядка (або немає newlines у рядку, на його початку чи в кінці тощо), можна використовувати символи ^ та \$ відповідно.
  - Якщо потрібно зіставити шаблон з усім рядком, добре включити обидва символи:
    - 'hello world' відповідає шаблону '^hello world\$'
    - 'hello worl' не відповідає шаблону '^hello world\$'
  - Символ «.» в регулярному виразі може зіставляти один будь-який символ.
    - Наприклад:
    - 'hello world' відповідає шаблону 'hel.o world'
    - 'helpo world' matches pattern 'hel.o world'
    - 'hel o world' matches pattern 'hel.o world'
    - 'helo world' не відповідає шаблону 'hel.o world'

# Matching a selection of characters

---

- Можна оточити набір символів квадратними дужками, щоб перевірити відповідність кожному з символів набору.
  - Якщо в регулярному виразі зустрічається `[abc]`, тільки один з цих символів зустрінеться в рядку пошуку, а, b або c.
- Кілька прикладів:
  - 'hello world' відповідає шаблону 'hel[lp]o world'
  - 'helpo world' matches pattern 'hel[lp]o world'
  - 'helPo world' не відповідає шаблону 'hel[lp]o world'
- Набори символів у квадратних дужках часто називають символьними класами (***character classes***).
  - Може виникнути потреба включити діапазон символів у набір.
  - Риска створить його в символьному наборі.
  - Дуже корисно для перевірки умов на зразок «усі маленькі літери», «усі літери», «усі числа»:
  - 'hello world' не відповідає шаблону 'hello [a-z] world'
  - 'hello b world' відповідає шаблону 'hello [a-z] world'
  - 'hello B world' matches pattern 'hello [a-zA-Z] world'
  - 'hello 2 world' matches pattern 'hello [a-zA-Z0-9] world'

# Управляючі символи (Escaping characters)

---

- Як перевірити наявність самої крапки в тексті?
  - Поширеним способом є додавання бекслешу перед символом, щоб зробити його управляючим.
  - Регулярний вираз для перевірки двоцифрового дробового числа в діапазоні від 0.00 до 0.99:
    - '0.05' відповідає шаблону '0\.[0-9][0-9]'
    - '005' не відповідає шаблону '0\.[0-9][0-9]'
    - '0,05' не відповідає шаблону '0\.[0-9][0-9]'
- Символи \[ та \ ( можна вставляти без створення символного класу.
  - Так же представляються спеціальні символи на зразок \n чи \t.
  - Деякі символні класи можна представити more succinctly за допомогою управляючих послідовностей: \s представляє пробільні символи, \w представляє літери, числа та «\_», \d - число:
  - '(abc]' відповідає шаблону '\(abc\]'
  - '1a' відповідає шаблону '\s\d\w'
  - '\t5n' не відповідає шаблону '\s\d\w'
  - '5n' відповідає шаблону '\s\d\w'

# Зіставлення багатьох символів

---

- Символ `*` вказує, що попередній шаблон можна зіставляти з 0 або більше символами.
  - Прості приклади:
    - `'hello' matches pattern 'hel*o'`
    - `'helo' matches pattern 'hel*o'`
    - `'hellllo' matches pattern 'hel*o'`
    - Решта символів (`h`, `e`, `o`) мають з'явитись точно один раз.
- Більш цікаво використовувати `*` для зіставлення з багатьма символами.
  - Наприклад, `.*` перевірятиме відповідність з будь-якими рядками, а `[a-z]*` лише колекції з маленьких літер, включаючи порожній рядок.
  - Наприклад:
    - `'A string.'` відповідає шаблону `'[A-Z][a-z]* [a-z]*\.'`
    - `'No .'` відповідає шаблону `'[A-Z][a-z]* [a-z]*\.'`
    - `"` відповідає шаблону `'[a-z]*.*'`

- 
- 
- Символ + поводитьсь аналогічно до зірочки: попередній шаблон може повторюватись один або більше разів, проте не опційно.
  - Символ ? перевіряє появу шаблону нуль або один раз.
  - Приклади (пам'ятайте, що \d відповідає тому ж символному класу, що і [0-9]):
    - '0.4' відповідає шаблону '\d+\.\d+'
    - '1.002' matches pattern '\d+\.\d+'
    - '1.' does not match pattern '\d+\.\d+'
    - '1%' matches pattern '\d?\d%'
    - '99%' matches pattern '\d?\d%'
    - '999%' не відповідає шаблону '\d?\d%'

# Що робити, якщо потрібні повторювані послідовності символів?

---

- Порівнювати такі шаблони:
  - 'abccc' відповідає шаблону 'abc{3}'
  - 'abccc' не відповідає шаблону '(abc){3}'
  - 'abcabcabc' відповідає шаблону '(abc){3}'
- Приклади регулярних виразів, які відповідають простим реченням латинкою:
  - 'Eat.' відповідає шаблону '[A-Z][a-z]\*([a-z]+)\*\.'
  - 'Eat more good food.' відповідає шаблону '[A-Z][a-z]\*([a-z]+)\*\.'
  - 'A good meal.' відповідає шаблону '[A-Z][a-z]\*([a-z]+)\*\.'

# Отримання інформації з регулярних виразів

---

- Модуль `re` постачає ОО інтерфейс, щоб працювати з регулярними виразами.
  - Функція `re.match()` повертає або не повертає `valid object`.
  - Якщо шаблон не підходить, повертається `None`.
  - Якщо є співпадіння, повертається корисний об'єкт, який можна інтроспектувати для інформації про шаблон.
- Часто цікаве питання типу «Якщо рядок відповідає шаблону, яке значення релевантного підрядка?»

```
pattern = "^[a-zA-Z.]+@[a-z.]*\.[a-z]+$"
search_string = "some.user@example.com"
match = re.match(pattern, search_string)

if match:
    domain = match.groups()[0]
    print(domain)
```

Метод `groups()` повертає кортеж усіх груп, що співпали всередині шаблону.

Групи впорядковані зліва направо.  
Групи можуть бути вкладеними.



# Корисні функції з модуля re: search() та findall()

---

- Функція `search()` знаходить перший випадок відповідності шаблону, послаблюючи обмеження на те, що шаблон порівнюється з початку рядка.
  - Аналогічного ефекту можна добитись, вводячи `^.*` на початку (front) шаблону, щоб зіставляти будь-які символи від початку рядка and the pattern you are looking for.
- Функція `findall()` поводитьсь схоже до `search()`, проте знаходить усі неперетинні входження в шаблоні для зіставлення.
  - Знаходить перше співпадіння, потім скидує пошук до кінця цього matching-рядка та знаходить наступну.
  - Повертає список або кортеж зіставлених рядків.
  - API поганий, інколи список, інколи кортеж: необхідно пам'ятати відмінності.
- Тип значення, яке повертається, залежить від кількості дужкових (bracketed) груп всередині регулярного виразу:
  - Якщо груп немає в шаблоні, `re.findall()` поверне список рядків, кожен елемент списку є повним підрядком початкового рядка, який відповідає шаблону
  - Якщо група відповідно до шаблону рівно одна, `re.findall()` поверне список рядків, кожен елемент списку – вміст групи.
  - Якщо в шаблоні кілька груп, `re.findall()` поверне список кортежів, кожен елемент кортежу містить значення з matching group. , in order

# Список чи кортеж?

---

- `>>> import re`
- `>>> re.findall('a.', 'abacadeafagah')`  
`['ab', 'ac', 'ad', 'ag', 'ah']`
- `>>> re.findall('a(.)', 'abacadeafagah')`  
`['b', 'c', 'd', 'g', 'h']`
- `>>> re.findall('(a)(.)', 'abacadeafagah')`  
`[('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'g'), ('a', 'h')]`
- `>>> re.findall('((a)(.))', 'abacadeafagah')`  
`[('ab', 'a', 'b'), ('ac', 'a', 'c'), ('ad', 'a', 'd'), ('ag', 'a', 'g'), ('ah', 'a', 'h')]`



# ДЯКУЮ ЗА УВАГУ!

Наступне питання: основи файлового вводу-виводу в Python