7. РОБОТА З ДАНИМИ В РІЗНИХ ФОРМАТАХ.

7.1. XML.

XML (Extensible Markup Language, розширювана мова розмітки) дозволяє налагоджувати взаємодію між програмами різних виробників, зберігати та обробляти складноструктуровані дані.

Мова XML (як і html) є підмножиною SGML, але її застосування не обмежені системою www. У XML можна створювати власні набори тегів для конкретної предметної області. У XML можна зберігати й обробляти бази даних та знань, протоколи взаємодії між об'єктами, опису ресурсів і багато чого іншого.

Перевага XML полягає в тому, що разом із даними вона зберігає й контекстну інформацію: теги та їхні атрибути мають імена.

Для подання букв та інших символів XML використовує Unicode, що зменшує проблеми з поданням символів різних алфавітів.

Наступний приклад достатньо простого XML-документа дає уявлення про цей формат (файл expression.xml).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<expression>
<operation type="+">
<operand>2</operand>
<operand>
<operation type="*">
<operand>3</operand>
<operand>4</operand>
</operation>
</operand>
</operation>
</expression>
```

XML-документ завжди має структуру дерева, у корені якого сам документ. Його частини, описувані вкладеними парами тегів, утворюють вузли. Таким чином, ребра дерева позначають "безпосереднє вкладення". Атрибути тегу вважають листками, як і найглибше вкладені частини, що не мають у своєму складі інших частин. Бачимо, що документ має деревоподібну структуру.

На відміну від html, у XML одиночні (непарні) теги записують із косою рискою:
, а атрибути – у лапках. У XML у назвах тегів та атрибутів має значення регістр літер.

7.1.1. Формування XML-документа

Концептуально є два шляхи обробки XML-документа: послідовна обробка та робота з об'єктною моделлю документа.

У першому випадку звичайно використовується SAX (Simple API for XML, простий програмний інтерфейс для XML). Робота SAX полягає в читанні джерел даних (input source) XML аналізаторами (XML-reader) та генерації послідовності подій (events), які обробляються об'єктами-обробниками (handlers). SAX надає послідовний доступ до XML-документа.

В іншому разі аналізатор XML будує DOM (Document Object Model, об'єктна модель документа), пропонуючи для XML документа конкретну об'єктну модель. У рамках цієї моделі вузли DOM-дерева доступні для довільного доступу, а для переходів між вузлами передбачено ряд методів.

Можна використовувати обидва ці підходи для формування наведеного вище XML-документа.

За допомогою SAX документ сформується як показано у прикладі:

```
import sys
from xml.sax.saxutils import XMLGenerator
g = XMLGenerator(sys.stdout)
g.startDocument()
g.startElement("expression", {})
g.startElement("operation", {"type": "+"})
g.startElement("operand", {})
g.characters("2")
g.endElement("operand")
g.startElement("operand", {})
g.startElement("operation", {"type": "*"})
g.startElement("operand", {})
g.characters("3")
g.endElement("operand")
g.startElement("operand", {})
g.characters("4")
g.endElement("operand")
g.endElement("operation")
g.endElement("operand")
g.endElement("operation")
g.endElement("expression")
g.endDocument()
```

Побудова дерева об'єктної моделі документа може мати вигляд як у прикладі наведеному нижче:

```
from xml.dom import minidom
dom = minidom.Document()
e1 = dom.createElement("expression")
dom.appendChild(e1)
p1 = dom.createElement("operation")
p1.setAttribute('type', '+')
x1 = dom.createElement("operand")
x1.appendChild(dom.createTextNode("2"))
p1.appendChild(x1)
e1.appendChild(p1)
p2 = dom.createElement("operation")
p2.setAttribute('type', '*')
x2 = dom.createElement("operand")
x2.appendChild(dom.createTextNode("3"))
p2.appendChild(x2)
x3 = dom.createElement("operand")
x3.appendChild(dom.createTextNode("4"))
p2.appendChild(x3)
x4 = dom.createElement("operand")
x4.appendChild(p2)
p1.appendChild(x4)
print (dom.toprettyxml())
```

Легко помітити, що при використанні SAX команди на генерацію тегів та інших частин видаються послідовно, а ось побудову однієї й тієї самої DOM можна виконувати різними послідовностями команд щодо формування вузла та його з'єднання з іншими вузлами.

7.1.2. Аналіз XML-документа

Для роботи з готовим XML-документом потрібно скористатися XML-аналізаторами. Аналіз XML-документа зі створенням об'єкта класу Document відбувається всього в одному рядку, за допомогою функції *parse()*. В даному розділі розглядається стандартна бібліотека *xml.etree.ElementTree*. Розглянемо тестовий xml-файл, який має наступний вигляд:

```
<?xml version="1.0"?>
<data>
```

```
    <country name="Liechtenstein">
      <rank>1</rank>
      <year>2008</year>
      <gdppc>141100</gdppc>
      <neighbor name="Austria" direction="E"/>
      <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
      <rank>4</rank>
      <year>2011</year>
      <gdppc>59900</gdppc>
      <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
      <rank>68</rank>
      <year>2011</year>
      <gdppc>13600</gdppc>
      <neighbor name="Costa Rica" direction="W"/>
      <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Імпортуємо бібліотеку *ElementTree*.

```
>>> import xml.etree.ElementTree as etree
```

Бібліотека *ElementTree* - частина стандартної бібліотеки Python, в пакеті xml.etree.ElementTree.

```
>>> tree = etree.parse('examples/feed.xml')
```

Основною точкою входу в бібліотеку ElementTree є функція parse(), яка приймає ім'я файлу, або потоковий об'єкт. Функція парсить весь документ за раз.

```
>>> root = tree.getroot()
```

Функція *parse()* повертає об'єкт що являє собою ввесь документ. Це не кореневий елемент.

Також можливо парсити дані використовуючи строкову змінну за допомогою функції *fromstring()*.

>>> root = etree.fromstring('xmlstring')

Для того щоб отримати посилання на кореневий елемент - викличте метод *getroot()*.

>>> root
<Element 'data' at 0x02EB60F0>

Змінна *root має* властивості tag та attrib, які виводять атрибути та їх значення для кожного тегу:

```
for child in root:
    print(child.tag, child.attrib)
```

```
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Для того, щоб отримати доступ для дочірніх елементів слід використовувати індекси:

>>> root[0][1].text
'2008'

Для того, щоб рекурсивно вивести всі дочірні елементи використовують функцію *iter():*

>>> for neighbor in root.iter('neighbor'): print(neighbor.attrib)

```
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

findall() знаходить тільки елементи з тегом, які є прямими нащадками поточного елемента. *find()* знаходить перший дочірній елемент з певним тегом. Функція *text* отримує доступ до текстового вмісту елемента. *get()* отримує доступ до атрибутів елемента:

```
>>> for country in root.findall('country'):
        rank = country.find('rank').text
        name = country.get('name')
        print(name, rank)
```

```
Liechtenstein 1
Singapore 4
Panama 68
```

7.2. Формат CSV.

Використання CSV (comma-separated values – значення, розділені комами) є поширеним способом переміщення даних в додатки типу електронних таблиць і з таких додатків з використанням звичайного тексту. Вміст файлів CSV – це лише ряди строкових значень, розділених комами.

На перший погляд, синтаксичний аналіз в цьому випадку повинен виявитися вельми простим. Досить було б, наприклад, регулярно виконувати *str.split(',')*.

Розглянемо короткий приклад запису CSV-даних в файл і подальше зчитування з нього.

```
import csv

data = (
   (9,'Web Clients and Servers','base64, urllib'),
   (10,'Web Progranrning : CGI & WSGI', 'cgi, time, wsgiref' ),
   (13 , 'Web Services ' , 'urllib, twython'),
   )

print(' *** WRITING CSV DATA ')
f = open('bookdata.csv' , 'w', newline='')
writer = csv.writer(f)
for record in data:
   writer.writerow(record)
f.close( )

print( ' ** * REVIEW OF SAVED DATA ' )
f = open ( 'bookdata.csv' , 'r' )
reader = csv.reader(f)
```

```
for chap, title, modpkgs in reader :
   print ('Chapter %s : %r ( featuring % s )' % (chap, title, modpkgs ) )

f.close( )
```

Результат виконання даного коду наведено нижче:

```
*** WRITING CSV DATA
 ** * REVIEW OF SAVED DATA
Chapter 9 : 'Web Clients and Servers' ( featuring base64, urllib )
Chapter 10 : 'Web Progranrning : CGI & WSGI' ( featuring cgi, time, wsgiref )
Chapter 13 : 'Web Services ' ( featuring urllib, twython )
```

Спочатку ми імпортуємо модуль *csv*. Слідом за оператором імпорту розташований наш набір даних *data*, який складається з потрійних кортежів.

Функція *csv.writer()* отримує на вході об'єкт типу "*відкритий файл*" (або файлоподібний об'єкт) і повертає об'єкт, що виконує запис (записуючий пристрій). Пристрій запису використовує метод *writerow()*, який дозволяє записувати в певний файл рядки даних, що розділяються комами. Після виконання запису цей файл закривається.

Функція *csv.reader()* протилежна до *csv.writer()*: вона повертає ітеруємий об'єкт, який можна використовувати для зчитування і виконання синтаксичного аналізу кожного рядка CSV-даних. Подібно *csv.writer()*, функція *csv.reader()* також отримує на вході об'єкт типу "відкритий файл" і повертає об'єкт, що виконує читання (пристрій читання).

7.3. Формат JSON.

JSON (JavaScript Object Notation) - простий формат обміну даними, заснований на підмножині синтаксису JavaScript. Модуль json дозволяє кодувати і декодувати дані в зручному форматі.

Приклад кодування основних об'єктів Python в JSON:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
```

```
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

Далі наведено приклад компактного кодування:

```
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'

>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
   "4": 5,
   "6": 7
}
```

Приклад декодування (парсинг) JSON-об’єктів в Python-структуру:

```
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
```

7.3.1. Основні методи

json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False) – серіалізує *obj* як форматований JSON потік в *fp*.

Якщо *skipkeys = True*, то ключі словника, які не входять до базового типу (*str, unicode, int, long, float, bool, None*) будуть проігноровані, замість того, щоб викликати виключення *TypeError*.

Якщо параметр *ensure_ascii* приймає значення *True*, всі не-ASCII символи у виводі будуть екрановані послідовностями \ *uXXXX*, і результатом буде рядок, що містить тільки ASCII символи. Якщо *ensure_ascii = False*, рядки запишуться як є.

Якщо параметр *check_circular* приймає значення *False*, то перевірка циклічних посилань буде пропущена, а такі посилання будуть викликати *OverflowError*.

Якщо *allow_nan = False*, при спробі серіалізовати значення дробних чисел, що виходять за допустимі межі, буде викликатися *ValueError (nan, inf, -inf)* в суворій відповідності зі специфікацією JSON, замість того, щоб використовувати еквіваленти

з *JavaScript (NaN, Infinity, -Infinity).*

Якщо *indent* є невід'ємним числом, то масиви і об'єкти в JSON будуть виводитися з цим рівнем вкладеності (доступу). Якщо рівень відступу 0, негативний або "", то замість цього просто використовуватимуться нові рядки. Значення за замовчуванням *None* відображає найбільш компактний вигляд.

При *sort_keys = True* словник буде відсортовано по ключам.

json.dumps (obj, skipkeys = False, ensure_ascii = True, check_circular = True, allow_nan = True, cls = None, indent = None, separators = None, default = None, sort_keys = False) - серіалізує obj в рядок JSON-формату. Аргументи мають те ж значення, що і для *dump()*.

Ключі в парах ключ/значення в JSON завжди є рядками. Коли словник конвертується в JSON, всі ключі словника перетворюються в рядки. В результаті цього, якщо словник спочатку перетворити в JSON, а потім назад в словник, то можна не отримати словник, ідентичний вихідному.

json.load (fp, cls = None, object_hook = None, parse_float = None, parse_int = None, parse_constant = None, object_pairs_hook = None) - перетворює JSON-дані в Python-структуру.

object_hook - опціональна функція, яка застосовується до результату декодування об'єкта (dict).

object_pairs_hook - опціональна функція, яка застосовується до результату декодування об'єкта з певною послідовністю пар ключ/значення.

Параметр *parse_float* застосовується для кожного значення JSON з плаваючою точкою. За замовчуванням, це еквівалентно float (num_str).

Параметр *parse_int*, застосовується для рядка JSON з числовим значенням. За замовчуванням еквівалентно int (num_str).

Параметр *parse_constant* застосовується для наступних рядків: "-Infinity", "Infinity", "NaN". Може бути використано для порушення винятків при виявленні помилкових чисел JSON.

Якщо не вдасться декодувати JSON, буде порушено виняток *ValueError*.

json.loads (s, encoding = None, cls = None, object_hook = None, parse_float = None, parse_int = None, parse_constant = None, object_pairs_hook = None) - декодує s (екземпляр str, що містить документ JSON) в об'єкт Python.

Інші аргументи аналогічні аргументам в load().

7.3.2. Класи модуля JSON

Клас *json.JSONDecoder(object_hook = None, parse_float = None, parse_int=None, parse_constant = None, strict = True, object_pairs_hook = None)* - простий декодер JSON. Виконує наступні перетворення при декодуванні:

Таблиця 7.1 – Декодування JSON-типів в Python-типи

JSON	**Python**
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Він також розуміє NaN, Infinity та Infinity як відповідні значення float, які знаходяться за межами специфікації JSON.

Клас *json.JSONEncoder (skipkeys = False, ensure_ascii = True, check_circular = True, allow_nan = True, sort_keys = False, indent = None, separators = None, default = None)* – кодує структуру даних Python в JSON-об’єкт.Він підтримує наступні об'єкти і типи даних, які вказано в Табл.7.2:

Таблиця 7.2 – Кодування Python-типів в JSON

Python	**JSON**
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null