РОБОТА З СЕРІАЛІЗАЦІЙНИМ ПРЕДСТАВЛЕННЯМ CSV

Питання 10.5.

Модуль csv

- Модуль csv кодує та декодує прості екземпляри списку або словника в CSV-нотацію.
  - Аналогічно до модуля json, це не зовсім повне рішення для персистентного зберігання.
  - Проте поширення CSV-файлів означає, що часто стає потрібною конвертація між Python-об'єктами та CSV.
- Робота з CSV-файлами включає ручне відображення об'єктів у CSV-структури.
  - Потрібно ретельно спроєктувати відображення, маючи на увазі обмеження CSV-нотації.
  - Це може бути складно через невідповідність вираження об'єктів і табличної структури CSV-файлу.
  - Вміст кожного стовпця CSV-файлу – текст.
- При завантаженні даних з CSV-файлу необхідно конвертувати ці значення в більш корисні типи в додатках.
  - Перетворення може ускладнитись через спосіб зведення типів у таблицях (spreadsheet). Наприклад, можемо мати spreadsheet, де поштові індекси США змінюються на дробові числа табличним процесором.
  - Коли таблиця зберігається в CSV, поштові індекси можуть стати дивними числовими значеннями, тому можливо застосувати перетворення на зразок ('ooooo'+row['zip']) [-5:] для відновлення провідних нулів.
  - Інший сценарій: "{0:05.0f}".format(float(row['zip'])) для відновлення провідних нулів.
  - Також не забувайте, що файл може мати суміш поштових індексів ZIP та ZIP+4.

Модуль csv

- Подальше ускладнення роботи з CSV-файлами – їх часте заповнення вручну та з помилками чи скороченнями.
  - Маючи відносно прості класи, можемо перетворити кожний instance у прості рядки (flat row) даних.
  - Часто namedtuple – хороший відповідник для CSV-файлу та Python-об'єктів.
  - Альтернатива: спроєктувати наші Python-класи навколо іменованих кортежів, якщо додаток зберігатиме дані в CSV-нотації.
- Для класів-контейнерів часто важко визначити представлення структурованих контейнерів у рядках CSVфайлів.
  - Хорошого вирішення таких невідповідностей немає, потрібно ретельно проєктувати перетворення даних.

Запис (Dumping) простих послідовностей у CSV

- Розглянемо відображення між екземплярами namedtuple та рядками в CSV-файлі на прикладі Python-класу:

```
from collections import namedtuple
GameStat = namedtuple( "GameStat", "player,bet,rounds,final" )
```

  - Об’єкти визначимо як просту послідовність атрибутів.
  - Продовжимо приклад з картами (класи Card, Hand, BlackJackHand). Заповнення таких об’єктів може виглядати так:

```
def gamestat_iter( player, betting, limit=100 ):
    for sample in range(30):
        b = Blackjack( player(), betting() )
        b.until_broke_or_rounds(limit)
        yield GameStat( player.__name__, betting.__name__, b.rounds,
b.betting.stake )
```

Запишемо дані в файл для подальшого аналізу

```
import csv
with open("blackjack.stats","w",newline="") as target:
    writer= csv.DictWriter( target, GameStat._fields )
    writer.writeheader()
    for gamestat in gamestat_iter( Player_Strategy_1, Martingale_Bet
):
        writer.writerow( gamestat._asdict() )
```

- Існує 3 кроки для створення CSV writer:

  1. Відкрити файл, задавши параметр newline = "" (можлива підтримка нестандартного завершення рядків для CSV-файлів).

  2. Створити CSV writer-об'єкт (тут – екземпляр DictWriter).

  3. Поставити заголовок на перший рядок файлу, що трохи спростить обмін даними, надаючи підказку про вміст CSV-файлу.

Запишемо дані в файл для подальшого аналізу

- Як тільки writer-об’єкт підготовано, застосуємо його метод writerow() для запису кожного словника в CSV-файл.
  - Деяке спрощення може дати метод writerows(), який очікує ітератор, а не окремий запис (row):

```
data = gamestat_iter( Player_Strategy_1, Martingale_Bet )
with open("blackjack.stats","w",newline="") as target:
    writer= csv.DictWriter( target, GameStat._fields )
    writer.writeheader()
    writer.writerows( g._asdict() for g in data )
```

  - Для методу writerows() отримаємо словник з кожного запису, отриманого від ітератора.

Завантаження (loading) простих послідовностей з CSV-файлу

- Можемо завантажувати прості послідовні об'єкти з CSV-файлу за допомогою циклу:

```
with open("blackjack.stats","r",newline="") as source:
    reader= csv.DictReader( source )
    for gs in ( GameStat(**r) for r in reader ):
        print( gs )
```

- Будо визначено об'єкт reader для файлу.
  - Оскільки відомо про коректний заголовок у файлі, можемо застосувати DictReader: перший рядок визначатиме назви атрибутів.
  - Далі конструюються об'єкти GameStat з рядків CSV-файлу за допомогою генераторного виразу (generator expression – розглядатимемо в темі про функціональне програмування).
- Тут припускаємо, що назви стовпців відповідають назвам атрибутів класу GameStat.
  - За потреби можемо підтвердити відповідність очікуваному формату, порівнюючи reader.fieldnames з GameStat._fields:
  - `assert set(reader.fieldnames) == set(GameStat._fields)`

Завантаження (loading) простих послідовностей з CSV-файлу

- Ми ігнорували типи даних зчитаних з файлу значень.
  - Значення двох числових полів таблиці при зчитуванні стануть рядками, тому потрібне складніше перетворення. Типовий вигляд такий:

```
def gamestat_iter(iterator):
    for row in iterator:
        yield GameStat( row['player'], row['bet'], int(row['rounds']),
int(row['final']) )
```

  - Іноді, коли файл має коректну шапку таблиці, проте некоректно введені дані, у функції int() може виникати ValueError.

- Кінцева версія з використанням генераторної функції:

```
with open("blackjack.stats","r",newline="") as source:
    reader= csv.DictReader( source )
    assert set(reader.fieldnames) == set(GameStat._fields)
    for gs in gamestat_iter(reader):
        print( gs )
```

Обробка контейнерів та складних класів

- Повертаючись до прикладу з мікроблогом, маємо Blog-об’єкт, який містить багато екземплярів класу Post.
  - При роботі з CSV-представленням потрібно спроєктувати відображення складних структур у табличний вигляд.
- Маємо 3 поширених вирішення:
  - ***Створювати 2 файли***: з блогами та постами. Файл blog.csv міститиме дані тільки екземплярів класу Blog без постів, а кожний рядок-пост з другого файлу посилатиметься на відповідний рядок з файлу blog.csv. Потрібно додавати ключ до кожного блогу, а кожний пост матиме зовнішній ключ, що посилається на ключ Blog-запису.
  - ***Створювати 2 види рядків у одному файлі***. Матимемо Blog-рядки та Post-рядки. Об’єкти-записувачі зав’язуються на різних типах даних, а об’єкти-зчитувачі відв’язують дані від їх типів.
  - ***Виконати реляційну операцію для баз даних join()*** між різними видами рядків, повторюючи інформацію щодо блогу для всіх його дочірніх постів.
- Найкращого вирішення серед представлених немає.
  - Проєктувати вирішення слід згідно з невідповідностями між CSV-рядками та більш структурованими Python-об’єктами.

Обробка контейнерів та складних класів

- Представлені способи мають свої переваги та недоліки.
  - Створення 2 файлів потребує спеціальних унікальних ідентифікаторів для кожного блогу та посту.
  - Не можна використовувати внутрішні ідентифікатори Python-об'єктів, оскільки не гарантується їх узгодженість після нових запусків.
- Загальне припущення: заголовок блогу – унікальний ключ, ***природний первинний ключ (natural primary key)***.
  - Проте добре працює такий підхід нечасто: ми не можемо змінити заголовок блогу без оновлення всіх постів, що належать цьому блогу.
  - Краще ввести унікальний ідентифікатор в класі. Такий ідентифікатор називають ***сурогатним ключем (surrogate key).***
  - Модуль Python uuid може постачати унікальні ідентифікатори для цього.

Запис (Dumping) та завантаження (loading) багатьох типів рядків у CSV-файл

- Створення кількох видів рядків у межах одного файлу дещо ускладнює формат.
  - Заголовки стовпців повинні стати об'єднанням (union) усіх доступних заголовків.
  - Оскільки можливість конфлікту назв у рядках різного виду присутня, можемо отримувати доступ до рядків за позицією (уникаючи використання csv.DictReader) або необхідно винайти більш складні заголовки, що комбінують назви класу та атрибутів.
- Простіше мати додатковий стовпчик і дописувати в кожний рядок значення, що уточнюватиме клас (дискримінатор рядка, row descriminator).
  - Назва класу відповідного об'єкта має добре спрацювати.
- Можемо записувати блоги та пости в один CSV-файл за допомогою 2 різних форматів запису рядків:

```
with open("blog.csv","w",newline="") as target:
        wtr.writerow(['__class__','title','date','title','rst_text','tags'])
        wtr= csv.writer( target )
        for b in blogs:
                wtr.writerow(['Blog',b.title,None,None,None,None])
                for p in b.entries:
                        wtr.writerow(['Post',None,p.date,p.title,p.rst_text,p.tags])
```

Запис (Dumping) та завантаження (loading) багатьох типів рядків у CSV-файл

```
with open("blog.csv","r",newline="") as source:
    rdr= csv.reader( source )
    header= next(rdr)
    assert header == ['__class__','title','date','title','rst_
text','tags']
    blogs = []
    for r in rdr:
        if r[0] == 'Blog':
            blog= Blog( *r[1:2] )
            blogs.append( blog )
        if r[0] == 'Post':
            post= post_builder( r )
            blogs[-1].append( post )
```

- Перетворення окремих типів даних дещо спантеличуватиме. Зокрема, були проігноровані типи даних полів timestamp та tags.
  - Можемо перезібрати об'єкти класів Blog і Post, переглядаючи їх дискримінатори рядка.
- Використовувались 2 припущення щодо стовпців CSV-файлу, які мають однаковий порядковий номер і тип, що і параметри конструкторів класів.
  - Для Blog-об'єктів використовувався blog= Blog(*r[1:2]), оскільки єдиний стовпчик є текстом, що відповідає конструктору класу.
  - При роботі із ззовні переданими даними дане припущення не спрацює.

Запис (Dumping) та завантаження (loading) багатьох типів рядків у CSV-файл

- Для збирання екземплярів класу Post використовується окрема функція, що відображає стовпці на конструктор класу:

```
import ast
def builder( row ):
    return Post(
        date=datetime.datetime.strptime(row[2], "%Y-%m-%d %H:%M:%S"),
        title=row[3],
        rst_text=row[4],
        tags=ast.literal_eval(row[5]) )
```

  - Перевагою є явність цього відображення.
  - Метод ast.literal_eval() застосовується для декодування більш складних літералів Python.
  - Це дозволяє містити в CSV-даних кортеж рядкових значень: "('#RedRanger', '#Whitby42', '#ICW')".

Запис (Dumping) та завантаження (loading) багатьох типів рядків у CSV-файл

```
def blog_iter(source):
    rdr= csv.reader( source )
    header= next(rdr)
    assert header == ['__class__','title','date','title','rst_
text','tags']
    blog= None
    for r in rdr:
        if r[0] == 'Blog':
            if blog:
                yield blog
            blog= Blog( *r[1:2] )
        if r[0] == 'Post':
            post= post_builder( r )
            blog.append( post )
    if blog:
        yield blog
```

- Виконаємо рефакторинг попередньої версії завантаження даних, щоб мати змогу ітерувати по об'єктах класу Blog, а не конструювати їх список.
  - Це дозволить переглядати великий CSV-файл та локалізувати лише релевантні Blog- та Post-рядки.
  - Дана функція є генератором, який генерує кожний екземпляр класу Blog окремо
- Функція blog_iter() створює об'єкт класу Blog та додає Post-об'єкти.
  - При кожній появі нового заголовку блогу попередній блог вважається повним і може генеруватись.
  - Наприкінці фінальний блог теж потрібно генерувати.
- За потреби у великому списку блогів можемо застосувати код:

```
with open("blog.csv","r",newline="") as source:
    blogs= list( blog_iter(source) )
```

Запис (Dumping) та завантаження (loading) багатьох типів рядків у CSV-файл

- Можемо застосувати наступний код для окремої обробки кожного блогу, здійснюючи рендеринг з метою створення RST-файлів:

```
with open("blog.csv","r",newline="") as source:
    for b in blog_iter(source):
        with open(blog.title+'.rst','w') as rst_file:
            render( blog, rst_file )
```

  - Функція blog_iter() застосовувалась для зчитування кожного блогу, а потім відбувався рендеринг.

- Окремий процес може запустити rst2html.py для перетворення кожного блогу в HTML-розмітку.
  - Можна запросто додавати фільтр, щоб обробляти тільки вибрані блоги відповідно до умови рендерингу.

Запис та завантаження з'єднаних (joined) рядків у CSV-файл

- З'єднання об'єктів означає, що кожний рядок є дочірнім об'єктом, поєднаним з усіма його батьківськими об'єктами.
  - Це веде до повторень атрибутів батьківського об'єкта для кожного дочірнього об'єкта.
  - Коли є кілька рівнів контейнерів, це призведе до величезної кількості дубльованої інформації.
- Перевагою повторів є повноцінна незалежність кожного рядка від контексту, визначеного в рядках вище.
  - Немає потреби в дискримінаторі класу, оскільки батьківські значення повторюються для кожного дочірнього об'єкта.
  - Це добре працює для даних, структурованих у просту ієрархію, проте при більш складних відношеннях шаблон «батько-дитина» порушується.
- У даних прикладах теги для посту зібрані докупи в єдиний текстовий стовпчик.
  - Якщо спробувати розбити теги на окремі стовпці, вони стануть дочірніми об'єктами посту, тобто текст посту буде повторюватись для кожного тегу, що не є хорошою ідеєю!

Запис та завантаження з'єднаних (joined) рядків у CSV-файл

- Заголовки стовпців повинні стати об'єднанням (union) доступних заголовків.
  - Через можливість конфлікту назв заголовків будемо кваліфікувати назву стовпця ще й назвою класу.
  - Заголовки матимуть вигляд 'Blog.title' чи 'Post.title', що дозволить використовувати класи DictReader та DictWriter замість позиційних аргументів.
  - Проте такі кваліфіковані назви нетривіально зіставляються з назвами атрибутів в означенні класу, що дещо ускладнює парсинг заголовків стовпців:

```
with open("blog.csv","w",newline="") as target:
    wtr= csv.writer( target )
    wtr.writerow(['Blog.title','Post.date','Post.title', 'Post.
tags','Post.rst_text'])
    for b in blogs:
        for p in b.entries:
            wtr.writerow([b.title,p.date,p.title,p.tags,p.rst_text])
```

Запис та завантаження з'єднаних (joined) рядків у CSV-файл

```
def blog_iter2( source ):
    rdr= csv.DictReader( source )
    assert set(rdr.fieldnames) == set(['Blog.title','Post.date','Post.
title', 'Post.tags','Post.rst_text'])
    row= next(rdr)
    blog= Blog(row['Blog.title'])
    post= post_builder5( row )
    blog.append( post )
    for row in rdr:
        if row['Blog.title'] != blog.title:
            yield blog
            blog= Blog( row['Blog.title'] )
        post= post_builder5( row )
        blog.append( post )
    yield blog
```

```
import ast
def post_builder5( row ):
    return Post(
        date=datetime.datetime.strptime(
            row['Post.date'], "%Y-%m-%d %H:%M:%S"),
        title=row['Post.title'],
        rst_text=row['Post.rst_text'],
        tags=ast.literal_eval(row['Post.tags']) )
```

- У такому форматі кожний рядок тепер містить об'єднання атрибуту Blog з Post-атрибутами.
  - Спрощує життя й те, що немає потреби заповнювати невикористані комірки значеннями None.
  - Оскільки кожна назва стовпця унікальна, можемо застосувати і DictWriter.
- Перший рядок даних використовується для збирання Blog-об'єкта та першого посту в цьому блозі.
  - The invariant condition for the loop that follows assumes that there's a proper Blog object.
  - Наявність коректного екземпляра класу Blog значно спрощує логіку обробки.

ДЯКУЮ ЗА УВАГУ!

Наступна тема: Побудова простих графічних інтерфейсів та візуалізація даних

Можемо відрефакторити Blog builder, щоб виокремити функцію

- However, it's so small that adherence to the DRY principle seems a bit fussy.
  - Because the column titles match the parameter names, we might try to use something like the following code to build each object:

```
    def make_obj( row, class_=Post, prefix="Post" ):
        column_split = ( (k,)+tuple(k.split('.')) for k in row )
        kw_args = dict( (attr,row[key])
            for key,classname,attr in column_split if
classname==prefix )
        return class( **kw_args )
```

- We used two generator expressions here.
  - The frst generator expression splits the column names into the class and attribute and builds a 3-tuple with the full key, the class name, and the attribute name.
  - The second generator expression flters the class for the desired target class; it builds a sequence of 2-tuples with the attribute and value pairs that can be used to build a dictionary.

This doesn't handle the data conversion for Posts.

- The individual column mappings simply don't generalize well.
  - Adding lots of processing logic to this isn't very helpful when we compare it to the post_builder5() function.
- In the unlikely event that we have an empty fle—one with a header row but zero Blog entries—the initial row=next(rdr) function will raise a StopIteration exception.
  - As this generator function doesn't handle the exception, it will propagate to the loop that evaluated blog_iter2(); this loop will be terminated properly.