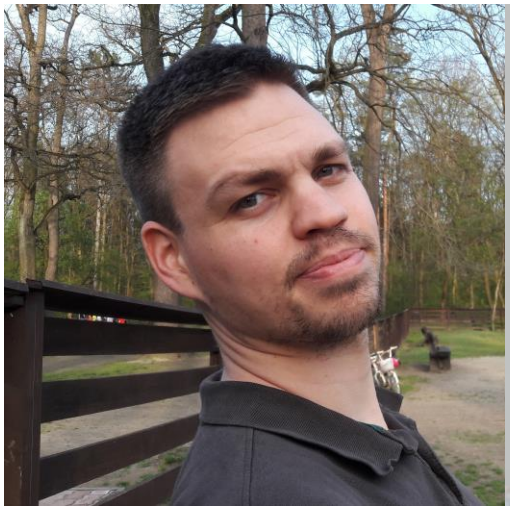


Dr. Hegedűs Péter, Dr. Ferenc Rudolf

Nagyméretű adatbázisok

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Data Wrangling Python és Pandas segítségével

Összefoglalás

Ez az olvasó lecke egy valós példákból összeállított, ám méreteiben lecsökkentett elképzelt használati eseten keresztül mutatja be a data wrangling folyamatát. Az olvasó képet kap arról, hogy mik azok a tipikus problémák, amiket egy ilyen adat transzformációs feladatnál le kell küzdeni, és gyakorlati tudást szerezhet a Python nyelv adatfeldolgozó képességéről, valamint a Pandas osztálykönyvtár alapvető használatáról. A leckékhez példa adatforrások, valamint az azokat feldolgozó szkriptek is társulnak.
A lecke fejezetei:

- 1. fejezet: **Egy tipikus adat transzformációs probléma leírása és adatforrások bemutatása (olvasó)**
- 2. fejezet: **A data wrangling feladat megfogalmazása és az alap funkciókkat tartalmazó megírása (olvasó)**
- 3. fejezet: **Komplexebb adatszűrést, transzformációt és formázást igénylő funckiók megvalósítása (olvasó)**

Téma típusa: **gyakorlati**
Olvasási idő: **50 perc**

📖 1. fejezet
Egy példa használati eset data wrangling alkalmazásához

Képzeljük el a következő helyzetet: adott egy (viszonylag nagy méretű) vállalkozás, ami ügyfeleknek értékesít valamilyen terméket. Ehhez több különböző részleg is kapcsolatban áll az ügyfelekkel, mint például a számlázó osztály, a ügyfélkapcsolati osztály, vagy éppen a sales részleg. Mindegyiküknek külön nyilvántartása van az ügyfelekről, más-más rendszereket/táblázatokat használnak, más-más adatokra fókuszálnak (pl. számlázási cím, ajánlatok küldése, személyes adatok, stb.). A vállalkozás menedzsmentje eldönti, hogy felhasználná az összes adatát, ami az ügyfelekről rendelkezésre áll abból a célból, hogy nyomon tudják követni kivel mikor vették fel utoljára a kapcsolatot, kit mikor milyen ajánlattal kell megkeresni, vagy éppen szükséges-e az ügyfelek adatainak frissítése. Ehhez azonban egységes formátumra kell hoznunk az összes adatot, és azokat egy egységes adattáblába szervezni, amin aztán kimutatások végezhetők.

A különböző részlegektől megkaptuk az általuk tárolt adatokat abban a formátumban, ahogy ők kényelmesen ki tudták exportálni. Így három fájlt kapunk feldolgozásra: két JSON [1] formátumú és CSV [2] az alábbi tartalmakkal.

Az ügyfélkapcsolati részleg elérhetővé tette az alábbi személyes adatokat tartalmazó JSON fájlt (`personal_entries.json`):

```
[
    {
        "PID": "1122334455",
        "name": "John Doe",
        "gender": "M",
```

```
        "last_contacted": "2017-03-02T11:43:00",
        "birth_year": 1987,
        "useless_info": [1, 2, 3]
    },
    {
        "PID": "2233445566",
        "name": "Jane Doe",
        "gender": "F",
        "last_contacted": "2019-10-12T15:22:34",
        "birth_year": 1965,
        "useless_info": [2, 3, 4]
    },
    {
        "PID": "3344556677",
        "name": "John Smith",
        "gender": "",
        "last_contacted": "2020-01-05T08:27:12",
        "birth_year": 1999,
        "useless_info": [1, 2, 3, 4, 5]
    }
]
```

A `PID` egy egyedi személyazonosító, a `last_contacted` pedig azt az időpontot adja meg, amikor az ügyfélkapcsolati osztály legutóbb kapcsolatba lépett az adott ügyféllel. A többi adat egyértelmű (a `useless_info` ignorálható, csak azért van ott, hogy látszódjon, nem mindig kell minden adatot felhasználnunk egy adatforrásból). Vegyük észre, hogy lehetnek hiányzó értékek a fájlban.

A számlázó osztály megküldte a maga részéről nyilvántartott számlázási címeket tartalmazo JSON fájlt (`billing_entries.json`):

```
[
    {
        "PID": 3344556677,
        "last_updated": 1578209232000,
        "address_info": {
            "city": "Minneapolis",
            "ZIP": 56112,
            "street": "Imaginary street",
            "number": 77
        }
    },
    {
        "PID": 1122334455,
        "last_updated": 1488451380000,
        "address_info": {
            "city": "Los Angeles",
            "ZIP": -1,
            "street": "Hollywood",
            "number": -1
        }
    },
    {
        "PID": 2233445566,
        "last_updated": 1519987380000,
        "address_info": {
            "city": "Miami",
            "ZIP": 65897,
```

```
            "street": "Dexter street",
            "number": 197
        }
    }
]
```

A `PID` ugyanaz az egyedi azonosító (vegyük észre, hogy itt számként adott, míg a másik fájlban string-ként), az `address_info` mező pedig az ügyfél számlázási címének részleteit írja le. A `last_updated` mező azt az időpontot adja meg, amikor az adott ügyfél adatait legutoljára egyeztették (epoch időformátumban, azaz a Unix epoch [1970-01-01] óta eltelt idő másodpercekben [3]). Hiányzó adatok itt is előfordulnak, amik explicit módon `-1`-es értékkel kerülnek feltöltésre.

Végezetül a sales osztály megosztotta velünk az általuk vezetett CSV fájlt, amiben az egyes ügyfelek ajánlattal történő megkereséseit vezetik (`sales_entries.csv`):

```
PID;offer_date;offer_text
3344556677;2020-01-05;"terrific offer"
1122334455;2017-03-02;""
2233445566;2019-10-12;
```

A `PID` a szokásos egyedi azonosító, ami után a legutóbbi ajánlat dátuma és a hozzá fűzött opcionális ajánlat szöveg következik.

📖 2. fejezet
A data wrangling feladat megfogalmazása és megoldása

A menedzsment felől érkező konkrét feladat így hangzik:

```
Készítsünk egy olyan Excel táblázatot (xlsx fájl) ezekből az adatokból, amiben
minden ügyfélhez pontosan egy sor tartozik, és az összes adat jelenjen meg egy-
egy oszlopban, amit a különböző részlegeknél nyilván tartunk róluk. A cím
adatoknál elég csak az irányító számot megjeleníteni, nem kell a város, utca és
házszám.
```

A megoldáshoz kézzel írunk egy data wrangling szkriptet, amihez a Python [4] programozási nyelvet használjuk majd, valamint a Pandas [5] adatfeldolgozó osztály könyvtárat. A program váza a következőképpen néz ki:

```python
import json
import pandas as pd
import os
import csv

SOURCE_1 = "personal_entries.json"
SOURCE_2 = "billing_entries.json"
SOURCE_3 = "sales_entries.csv"
OUTPUT   = "merged_data.xlsx"

# Data source loader functions

if __name__ == "__main__":
    df1 = load_src1(os.path.join("source", SOURCE_1))
    df2 = load_src2(os.path.join("source", SOURCE_2))
```

```
    df3 = load_src3(os.path.join("source", SOURCE_3))
    result = df1.merge(df2, on='PID').merge(df3, on='PID')
    result.to_excel(OUTPUT)
```

Betöltjük a szükséges modulokat és definiáljuk a három bemenő adat fájlunkat, valamint a kimeneti fájlt. A program végrehajtásakor végrehajtjuk a 3 adat betöltését egy-egy DataFrame-be, ami a Pandas könyvtár alapvető adatstruktúrája. Egy két dimenziós adattáblát implementál, amihez soronként és oszloponként is hozzá tudunk férni. Ezután kihasználjuk a Pandas könyvtár hatékony API-ját, hiszen a három különböző adatforrásból érkező adattáblát egyszerűen összefűzhetjük a `merge` metódus segítségével. Csak azt kell megadnunk, hogy egy `DataFrame`-hez melyik másik `DataFrame`-et szeretnénk hozzáfűzni, és azt, hogy melyik oszlop értékét használja a sorok összerendeléséhez. Szerencsénkre minden fájl tartalmazza a `PID` azonosítót, így ezeket felhasználva össze tudjuk fűzni az adatokat (a valóságban nincs mindig ekkora szerencsénk). Amint a memóriában előállt a végleges adattábla, egyetlen hívással kimenthetjük azt egy Excel táblába, lásd a `to_excel()` hívást, ami szintén nagyon kényelmes.

Töltsük ki a hiányzó részeket, adjuk hozzá az egyes adat betöltő függvényeket a szkriptünkhöz, amik az egyes különböző fájlokból elkészítik a megfelelő tartalmú `DataFrame`-et.

```
def load_src1(json_path):
    '''
    Process the personal entries data source
    '''
    entries = json.load(open(json_path, 'r'))
    entry_list = list()
    for entry in entries:
        row = list()
        row.append(entry["PID"])
        row.append(entry["name"])
        row.append(entry["gender"])
        row.append(entry["last_contacted"])
        row.append(entry["birth_year"])
        entry_list.append(row)

    return pd.DataFrame(entry_list, columns=
["PID","name","gender","last_contacted","birty_year"])
```

Az első függvény a személyes adatokat tölti be. A Python `json` modulja segít a JSON feldolgozásában, egy `load` egy Python objektumot készít a JSON tartalmából, így csak be kell járnunk az egyes bejegyzéseket, és minden JSON entry értékből egy Python listát készítünk. Ezeket a listákat (ami egy-egy sornak fognak megfelelni az adattáblánkban) egy `entry_list` nevű másik listában gyűjtjük össze. Végül visszaadunk egy új `DataFrame` objektumot, aminek az értékei az `entry_list` tartalmazza, míg a `columns` paraméter definiálja, hogy milyen fejlécet használjunk az adatokhoz.

```
def load_src2(json_path):
    '''
    Process the billing entries data source
    '''
    entries = json.load(open(json_path, 'r'))
    entry_list = list()
    for entry in entries:
        row = list()
        row.append(str(entry["PID"]))
```

```
        row.append(entry["last_updated"])
        row.append(entry["address_info"]["ZIP"])
        entry_list.append(row)

    return pd.DataFrame(entry_list, columns=["PID","last_updated","ZIP"])
```

A második JSON betöltése szinte teljesen megegyezik az előzővel, természetesen más mező neveket és fejléceket használ. Egyetlen dologra kell odafigyelnünk, hogy míg az előző JSON string-ként tárolta a `PID` értékeket, ez a JSON számként. Hogy később ezeket össze lehessen rendelni, a `str()` függvény segítségével itt is rögtön string-gé konvertáljuk már a `DataFrame` létrehozása előtt.

```
def load_src3(csv_path):
    '''
    Process the offer entries data source
    '''
    entries = csv.reader(open(csv_path, 'r'), delimiter=';', quotechar='"')
    entry_list = list()
    for entry in entries:
        row = list()
        row.append(entry[0])
        row.append(entry[1])
        row.append(entry[2])
        entry_list.append(row)

    return pd.DataFrame(entry_list, columns=["PID","offer_date","offer_text"])
```

Végezetül a CSV betöltéséhez a `csv` modult használjuk, ahol a sorokon végig iterálva ugyanúgy létrehozzuk a listák listáját, amiből a `DataFrame` elkészíthető. A CSV esetében az oszlopok sorszámaival tudunk hivatkozni az egyes sorokon belüli mezők értékeire.

Ezzel el is készült az adatfeldolgozó szkriptünk alap változata, amely a három adatforrást összefűzi és kimenti egy Excel adattáblába (lásd `wrangler_v1.py`). A szkript futtatása előtt a szükséges függőségeket az alábbi paranccsal telepíthetjük:

```
pip install -r requirements.txt
```

📖 3. fejezet
Komplexebb adatfeldolgozó funkciók hozzáadása

A kezdeti dicséret után további feladatokat kapunk, hogy még hasznosabb legyen az összeállított adattábla.

Hiányzó értékek és életkor kezelése

Explicit `NaN` értékekkel helyettesítsük a táblázatban az összes hiányzó értéket. Az üres cellák, és a `-1`-es mező értékek számítanak hiányzó értéknek. Ezek mellett kiderült, hogy a születési év mező helyett sokkal praktikusabb lenne az ügyfelek életkorát beírni a táblába, mert a későbbiek során korosztályonkénti statisztikát szeretne végezni a marketing csoport. Lássunk is neki a feladathoz módosítsuk a meglévő data wrangling szkriptet:

```
if __name__ == "__main__":
    df1 = load_src1(os.path.join("source", SOURCE_1))
    df2 = load_src2(os.path.join("source", SOURCE_2))
    df3 = load_src3(os.path.join("source", SOURCE_3))
    result = df1.merge(df2, on='PID').merge(df3, on='PID')
    # Replace empty values with explicit NaN
    for column in result:
        result[column].mask(result[column] == "", "NaN", inplace=True)
        result[column].mask(result[column] == "-1", "NaN", inplace=True)
    result.to_excel(OUTPUT)
```

A program belépési pontján miután előállítottuk az összefűzött `DataFrame`-et, oszloponként végigmegyünk rajta, és a `mask()` függvény segítségével bizonyos értékeket lecserélünk, amennyiben az megfelel egy feltételnek. Két feltételünk van, egyrészt az üres string-eket, másrészt a `-1`-es értékeket kell helyettesítenünk `NaN` értékkel. Ezt a Pandas segítségével a fenti néhány sorban kezelni tudjuk.

```
def _convert_to_age(birth_year):
    '''
    A helper function to calculate age based on birth year
    '''
    now = datetime.datetime.now()
    return now.year - birth_year

def load_src1(json_path):
    '''
    Process the personal entries data source
    '''
    entries = json.load(open(json_path, 'r'))
    entry_list = list()
    for entry in entries:
        row = list()
        row.append(entry["PID"])
        row.append(entry["name"])
        row.append(entry["gender"])
        row.append(entry["last_contacted"])
        # Convert birth year to current age
        row.append(_convert_to_age(entry["birth_year"]))
        entry_list.append(row)

    return pd.DataFrame(entry_list, columns=
["PID","name","gender","last_contacted","age"])
```

A születési év az első adatfájlból, a személyes adatokat tartalmazó JSON-ből jön, így az azt feldolgozó függvényt kell módosítanunk. Ahelyett, hogy közvetlenül eltárolnánk a beolvasott születési évet, meghívjuk rá a `_convert_to_age()` segédfüggvényt, ami aktuális életkorrá alakítja a születési évet. Ezt követően ezt a módosított értéket tároljuk el a `DataFrame`-ben `age` oszlop címkét használva. A konverzió pedig szintén egyszerű, a `datetime` standard Python modul segítségével (ne feledjük, hogy ezt is importálni kell) lekérjük az aktuális időt, amiből kiolvassuk az év értékét, majd az aktuális évből egyszerűen kivonjuk a születési évet, és visszaadjuk a számított életkort.

Adattábla oszlopainak módosítása

Újabb dicséret után ismételten feladatot kapunk. Szükség lenne egy újabb oszlopra, ami egy `True/False` flag-et tartalmaz, aszerint hogy szükséges-e felvenni a kapcsolatot az ügyféllel adategyeztetés céljából. Ezt a flag-et pedig úgy tudjuk beállítani, hogy amennyiben a `last_contacted` mező értéke frissebb, mint a `last_updated` mezőé, azaz a legutóbbi megkereséskor nem történt adategyeztetés is, akkor ezt a flag-et `True`-ra kell állítani, azaz szükséges az adategyeztetés, ha a két dátum egyezik, vagy a `last_updated` frissebb, akkor a flag `False`. Azonban a dátumoktól függetlenül ha az irányítószám értéke hiányzik, akkor minden esetben szükséges az adat frissítése. Az új oszlop létrehozása után a két dátum oszlop törölhető az adattáblából.

```
def reshape_df(df):
    '''
    Extend the dataframe with a new column called "update_needed"
    Its value is true when the last_contacted value is more recent than
last_updated
    Also remove the last_contacted and last_updated columns as they become
unnecessary
    '''

    def _compare_dates(row):
        # We need to convert the different date formats into a datetime object
        lc = datetime.datetime.strptime(row['last_contacted'], '%Y-%m-
%dT%H:%M:%S')
        lu = datetime.datetime.fromtimestamp(row['last_updated']/1000)
        return lc > lu or row['ZIP'] == "NaN"

    df['update_needed'] = df.apply(_compare_dates, axis=1)
    return df.drop(columns = ['last_contacted', 'last_updated'])

...

if __name__ == "__main__":
    df1 = load_src1(os.path.join("source", SOURCE_1))
    df2 = load_src2(os.path.join("source", SOURCE_2))
    df3 = load_src3(os.path.join("source", SOURCE_3))
    result = df1.merge(df2, on='PID').merge(df3, on='PID')
    # Replace empty values with explicit NaN
    for column in result:
        result[column].mask(result[column] == "", "NaN", inplace=True)
        result[column].mask(result[column] == "-1", "NaN", inplace=True)
    result = reshape_df(result)
    result.to_excel(OUTPUT)
```

Ehhez írunk egy újabb függvényt, ami egy új oszlopot ad a `DataFrame`-hez, valamint törli a két dátumos oszlopot. A `datetime` modul segítségével mindkét dátumot (string-ból és epoch timestamp-ből) azonos `datetime` objektummá alakítjuk a fenti hívások segítségével [6], majd az `apply()` függvény segítségével létrehozunk egy egész új oszlopot. Az `apply` a paraméterben megadott függvényt a `DataFrame` minden egyes sorára meghívja, és az általa visszaadott érték kerül az új oszlop adott sorába. Ezután a `drop` segítségével a megfelelő fejlécű oszlopokat egyszerűen eltávolítjuk. A program belépési pontján pedig a végső kiíratás előtt meghívjuk az újonnan létrehozott `reshape_df()` függvényt az előállt `DataFrame` objektumra.

További feladatok

1. Módosítsuk úgy a data wrangling szkriptet, hogy a végső adattáblába ne csak az irányítószám, hanem a teljes cím adatok is bekerüljenek egy-egy oszlopban! ⭐
2. A végső adattáblába vegyünk fel egy újabb oszlopot, ami szintén egy `True/False` flag-et tartalmazzon, méghozzá akkor legyen `True`, ha az `offer_date` mező értéke már 6 hónapnál régebbi dátumot tartalmaz (azaz az ajánlat lejárt), egyébként pedig az érték legyen `False`! ⭐
3. Oldjuk meg a fenti feladatot a Pandas könyvtár használata nélkül! ⭐⭐

Referenciák

[1] https://www.json.org/json-en.html

[2] https://en.wikipedia.org/wiki/Comma-separated_values

[3] https://en.wikipedia.org/wiki/Unix_time

[4] https://www.python.org/

[5] https://pandas.pydata.org/

[6] https://www.journaldev.com/23365/python-string-to-datetime-strptime