6. Generování náhodných čísel a testování generátorů

Zadání:

Tento úkol bude poněkud kreativnější charakteru. Vaším úkolem je vytvořit vlastní generátor semínka do pseudonáhodných algoritmů. Jazyk Python umí sbírat přes ovladače hardwarových zařízení různá fyzická a fyzikální data. Můžete i sbírat data z historie prohlížeče, snímání pohybu myší, vyzvání uživatele zadat náhodné úhozy do klávesnice a jiná unikátní data uživatelů.

Řešení:

Mým úkolem bylo vytvořit vlastní generátor pseudonáhodných čísel. Já jsem **nakonec zkonstruoval dva**.

První generátor:

Tento generátor je založen na procentuálním vytížení procesoru a aktuální timestamp v unixové formě. Jako první jsem vytvořil dvě funkce. První funkce date() mi vrací čas v unixové formě, se kterou následně počítám při výpočtu seedu. Pro zjištění času používám knihovnu datetime.

```
def date():
    current_datetime = datetime.datetime.now()
    timestamp = current_datetime.timestamp()
    print(timestamp)
    return timestamp
```

Před touto funkcí jsem si akorát naimportoval knihovny a zvolil si, kolika ciferné chci výsledná čísla.

```
import psutil
import datetime
import time

digits_to_keep = 5
```

Druhá hlavní funkce je seed(), která používá jako argument počet iterací. Následně v této funkci je for cyklus, který v počtu iterací provede záznam procentuálního zatížení v intervalu 0.1 ms, následně tuto hodnotu vloží do listu.

```
def seed(iter):
    # SEED
    cpu_list = []

    # cpu % v intervalu 0,1
    for i in range(iter):
        cpu = psutil.cpu_percent(interval=0.1)
        cpu_list.append(cpu)
        print("ve for cpu%usage", cpu_list)
```

Po dokončení sekvence zátěže CPU přijde na řadu načtení aktuálního clock time pomocí time.perf_counter(). Ten následně použiji při výpočtu seedu společně s předchozími funkcemi.

```
    cpu_clock = time.perf_counter()

    print("AKTUALNI CPU up time clock:",cpu_clock)

    cpu_perc = round(((sum(cpu_list) * cpu_clock)*date()))

    shorted_number = cpu_perc//10**(len(str(cpu_perc)) - digits_to_keep)

    print("Výsledná suma rounded:", shorted_number)
    return shorted_number
```

Při výstupu dochází ještě k formátování výsledného seedu, na začátku programu jsem si definoval velikost digits_to_keep na 5 cifer.

Samotný výpočet je jednoduchý, vezmu sumu z listu, ve kterém mám zaznamenané zatížení CPU, tuto sumu nakonec vynásobím clock time a následně i unixový formát datumu (ten je ve formátu počet sekund od nějakých 70. let).

Následně výsledek přeformátuji a vrátím si ho z funkce.

Úplně poslední funkcí je check(), která mi slouží k analýze získaných čísel. Před začátkem si vytvořím kopii výsledného seznamu. Ten mi slouží k zaznamenání postupu porovnání hodnot.

Ve funkci si přidávám duplicitní čísla do seznamu duplicit, který slouží pro záznam duplicitních čísel. Porovnávání funguje tak, že ve for cyklu procházím celý výsledný seznam a porovnávám ho s jeho kopií, při každé iteraci odstraním z kopie aktuální prvek. Následně se ptám, zda je další výskyt tohoto prvku v tomto seznamu, pokud ne, tak je vše v pořádku a jdu o iteraci dál. Pokud ale ne, tak došlo ke shodě a daný prvek přiřadím do seznamu duplicit.

Následně se ptám, kolik prvků je v seznamu duplicit. To je mým výsledkem celé metody.

```
def check(analytic_seed, analytic_seed_remover):
    duplicit = []
    for _ in range(len(analytic_seed)):
        print("Číslo pozice:", _)

        y = analytic_seed[_]
        print("y je:", y)

        analytic_seed_remover.remove(y)
        print("Zbylé znaky ke kontrole:", analytic_seed_remover)

        if y not in analytic_seed_remover:
            print("Ales gute!")
        else:
            print("Schoda na znaku!:",y)
            duplicit.append(y)
            next

    print("DUPLICITY:", duplicit)
    print("POČET DUPL. ZNAKŮ:", len(duplicit))
    return duplicit
```

Zde mám tělo programu, definuju si počet vygenerovaných čísel. Následně v daném for cyklu pouštím funkci seed(), ve které zaznamenávám na jednu rundu 15 procentuálních záznamů o zatížení cpu.

Následně celý výstup přidělím do listu, který budu pomocí funkce check() procházet.

```
# ANALÝZA
pocet_prvku = 1000
analytic_seed = []

for i in range(pocet_prvku):
    x = seed(15)
    analytic_seed.append(x)

print(analytic_seed)
print(len(analytic_seed))

analytic_seed_remover = analytic_seed.copy()
```

Na posledním řádku vytvářím již zmíněnou kopii výsledného listu.

```
#RUN CHECK
check(analytic_seed, analytic_seed_remover)
```

Můj úplně poslední krok je volání funkce check() pro kontrolu vygenerovaných čísel.

Výsledný výstup:

```
Číslo pozice: 997
y je: 35490
Zbylé znaky ke kontrole: [26561, 18346]
Ales gute!
Číslo pozice: 998
y je: 26561
Zbylé znaky ke kontrole: [18346]
Ales gute!
Číslo pozice: 999
y je: 18346
Zbylé znaky ke kontrole: []
Ales gute!
DUPLICITY: [15082, 30423, 39413, 27750, 23243, 17405, 20285, 25438]
POČET DUPL. ZNAKŮ: 8
```

Zde přikládám výstup vygenerovaných 1000 náhodných čísel o velikosti 5 bitů. Při rundě zátěže CPU jsem zaznamenával 15 hodnot. Nakonec jsem získal 8 duplicitních čísel, což není úplně skvělý výsledek.

```
ve for cpu%usage [2.7]
ve for cpu%usage [2.7, 0.0]
ve for cpu%usage [2.7, 0.0, 1.2]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6, 7.8]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6, 7.8, 17.8]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6, 7.8, 17.8, 14.3]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6, 7.8, 17.8, 14.3, 19.2]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6, 7.8, 17.8, 14.3, 19.2, 10.7]
ve for cpu%usage [2.7, 0.0, 1.2, 9.5, 0.0, 0.0, 1.3, 1.3, 2.6, 7.8, 17.8, 14.3, 19.2, 10.7, 0.0]
AKTUALNI CPU up time clock: 193222.225948
1685276812.293437
Výsledná suma rounded: 28785
```

Zde je zaznamenaný celý cyklus ve funkci seed().

Druhý generátor:

Mojí prvotní myšlenkou bylo to, že bych chtěl z Google Chrome historie vytáhnou data, které mi poslouží ke generování náhodných čísel. Bohužel se mi nepodařilo data scrapovat přímo z prohlížeče, proto jsem si pomocí pluginu vytáhl CSV soubor historie. Ta mi sloužila jako můj vstup.

```
order,id,date,time,title,url,visitCount,typedCount,transition
0,32626,5/11/2023,8:10:49,Export Chrome History - Internetový obchod Chrome,h
1,32625,5/11/2023,8:10:44,Export Chrome History - Internetový obchod Chrome,h
2,32623,5/11/2023,8:10:43,how to export my chrome history to html file - Hled
3,32624,5/11/2023,8:09:43,Can Chrome browser history be exported to an HTML f
4,32623,5/11/2023,8:09:41,how to export my chrome history to html file - Hled
5,32623,5/11/2023,8:09:40,how to export my chrome history to html file - Hled
6,32622,5/11/2023,8:09:11,Google - Moje aktivita,https://myactivity.google.co
7,32622,5/11/2023,8:09:10,Google - Moje aktivita,https://myactivity.google.co
8,32615,5/11/2023,8:05:26,whats the xpath for google chrome history names - H
9,32615,5/11/2023,8:05:26,whats the xpath for google chrome history names - H
10,32615,5/11/2023,8:00:36,whats the xpath for google chrome history names -
11,32615,5/11/2023,8:00:36,whats the xpath for google chrome history names -
12,32617,5/11/2023,7:59:48,How to find element by XPath in Selenium | Browser
13,32615,5/11/2023,7:59:45,whats the xpath for google chrome history names -
14,32614,5/11/2023,7:59:34,whats the xpath for google chrome history names -
15,32614,5/11/2023,7:59:33,whats the xpath for google chrome history names -
```

Můj první úkol byl přečíst data z CSV souboru. K tomu jsem využil knihovnu pandas a její funkci pandas.read_csv(). Po jejich přečtení jsem potřeboval nějak převést data na nějaké číselné hodnoty.

```python
# export historie do csv, přes rozšíření chromu, scraping nevyšel
data_set = pd.read_csv("history.csv")
```

Proto jsem se rozhodl pro vytvoření slovníku, ve kterém mají dané znaky přiřazenou svojí hodnotu.

```python
slovnik = {
    'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9, 'j': 10,
    'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16, 'q': 17, 'r': 18, 's': 19,
    't': 20, 'u': 21, 'v': 22, 'w': 23, 'x': 24, 'y': 25, 'z': 26,
    'A': 27, 'B': 28, 'C': 29, 'D': 30, 'E': 31, 'F': 32, 'G': 33, 'H': 34, 'I': 35, 'J': 36,
    'K': 37, 'L': 38, 'M': 39, 'N': 40, 'O': 41, 'P': 42, 'Q': 43, 'R': 44, 'S': 45,
    'T': 46, 'U': 47, 'V': 48, 'W': 49, 'X': 50, 'Y': 51, 'Z': 52,
    '0': 53, '1': 54, '2': 55, '3': 56, '4': 57, '5': 58, '6': 59, '7': 60, '8': 61, '9': 62,
    '!': 63, '@': 64, '#': 65, '$': 66, '%': 67, '^': 68, '&': 69, '*': 70, '(': 71, ')': 72,
    '-': 73, '_': 74, '=': 75, '+': 76, '[': 77, ']': 78, '{': 79, '}': 80, '|': 81, '\\': 82,
    ';': 83, ':': 84, '\'': 85, '"': 86, ',': 87, '.': 88, '/': 89, '<': 90, '>': 91, '?': 92,
    '`': 93, '~': 94
}
```

Následně jsem si přetypoval dané sloupce do jednotlivých listů, které slouží pro následnou filtraci a převod dat. Toto jsem provedl pomocí funkce tolist().

```
# PŘEVOD COLUMN -> DATALISTU -> TOLIST()
order = data_set['order'].tolist()
id = data_set['id'].tolist()
date = data_set['date'].tolist()

time = data_set['time'].tolist()
title = str(data_set['title'].tolist())
url = str(data_set['url'].tolist())
VisitCount = data_set['visitCount'].tolist()
TypedCount = data_set['typedCount'].tolist()
```

Pro generaci čísel jsem si zvolil pouze dva listy (title column a url column). Poté jsem každým cyklem prošel celý list a následně hodnoty převedl.

Převod title na číselné hodnoty:

```
# převod characters titlu z historie
title_num = []

counter = 0
for i in range(len(title)):
    counter += 1
    for ch in title[i]:
        if ch in slovnik and ch in title_num:
            dupl_char = slovnik[ch] + counter
            title_num.append(dupl_char)
        elif ch in slovnik:
            title_num.append(slovnik[ch])
        else:
            next


print("TITLE NUM:", title_num)
```

Zde procházím cyklem celým seznamem title pomocí pozice indexu, následně kontroluji, zda jsem daný znak již převedl nebo ne. Pokud se mi daný znak ch nachází ve slovníku a zároveň jsem ho už “odbavil”, tak do výsledného seznamu přidám hodnotu ze slovníku + hodnota counteru.

Pokud je ale pouze znak ch ve slovníku a ne již odbaven, tak k němu přiřadím pouze hodnotu ze slovníku. Ovšem pokud se mi znak neobjeví ve slovníku, tak ho přeskakuji a jdu dál.

Převod url na číselné hodnoty:

```
#TO SAME PRO URL
url_num = []

for i in range(len(url)):
    counter += 3
    for ch in url[i]:
        if ch in slovnik:
            dupl_char_url = slovnik[ch]
            url_num.append(dupl_char_url)
        elif ch in slovnik and ch in url_num:
            dupl_char_url = slovnik[ch] + counter
            url_num.append(dupl_char_url)
        else:
            next

print("URL NUM:", url_num)
```

To samé aplikuji na url adresu. Akorát jsem změnil hodnotu counteru. Podmínky jsou ale jinak prakticky nezměněné.

Poznámka ke generátoru:

Následující tělo programu bylo přímo na míru pro mojí historii prohlížeče. Kvůli časté duplicitě se mi potkávali hodnoty, které obsahovaly *999. Proto jsem u finálního výsledku použil spoustu podmínek s wild cards. abych jejich výskyt eliminoval.

Proto si myslím, že tento generátor nebude úplně použitelný pro další jinačí záznamy. Možná historie, která nemá tolik duplicit by fungovala dobře. Celkově následující část je celkem dost chaotická.

Tělo výpočtu seedu:

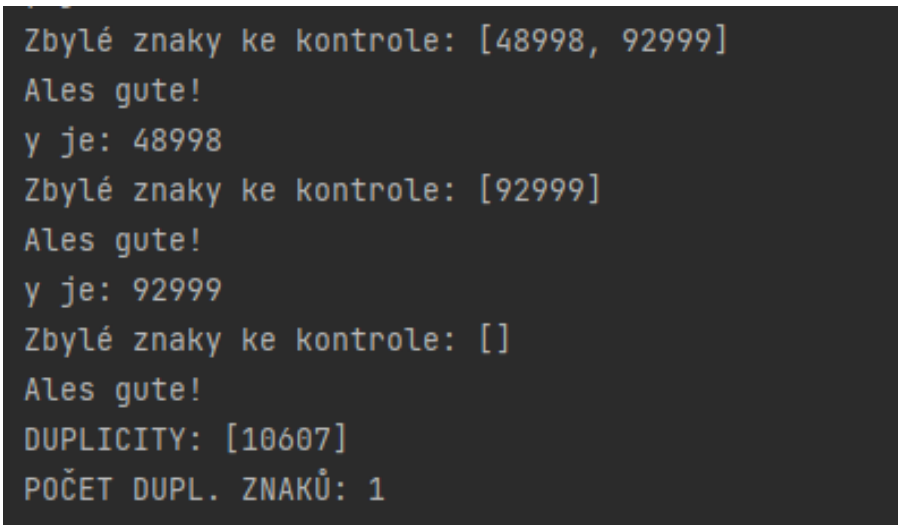
```python
#SEED GENERATOR
analytic_seed = []
pocet_cisel = 1000
digits_to_keep = 5

k = 0
for x in range(pocet_cisel):
    k += 1

    rn = title_num[x] + url_num[x]
    shorted_number = round((rn//10**(len(str(rn)) - digits_to_keep)))

    if re.search('.*99', str(shorted_number)) or re.search('.*999', str(shorted_number)):
        shorted_number += k
        analytic_seed.append(shorted_number)

    else:
        analytic_seed.append(shorted_number)
```

V cyklu z rozsahu počtu požadovaných čísel putuje přes hromadu podmínek, které slouží pro snížení duplicit.

Pokud je k % 2 == 1, tak můj výsledný seed je součet dvou hodnot z dvou listu na stejnem indexu. Pokud toto dane číslo projde podmínkou s wild card, tak dojde k přičtení k (pokud dané číslo končí na ***99 / **999). Pokud tak není, tak se formátované číslo na 5 bitů přejde do výsledného listu.

Pokud neprojde první podmínka, tak ve druhé je průběh dost podobný, akorát přičítám += k +1.

Výsledný výstup:



Konečná kontrola přes funkci check() jsem získal 1 duplicitu na 1000. Čísla byla naformátována na 5 cifer.

Závěr:

První generátor mi hodil duplicitu o 8 prvcích. Druhý generátor duplicitu o 1 prvku. Co se týče ale použitelnosti, tak první generátor je rozhodně více efektivní a použitelný. Co se týče časové náročnosti, tak první generátor je časově o dost náročnější, jelikož musí proběhnout celý cyklus záznamu využití CPU. Což u druhého generátoru nehrozí, zde se akorát čtou data z CSV souboru (můžeme vzít v potaz importování a generace CSV z chromu).

ODKAZ NA GITHUB:

https://github.com/Marty808s/MSW_6_Generator