Lucrarea 3

Elemente de Python - III

1. *Regular expressions în linia de comandă*

O expresie regulată (regexp) este o sintaxă care permite căutarea după un pattern (model, şir de căutare). Programul *grep* (*general regular expression parser*) permite căutarea simplă după șiruri, sau după *regexp*-uri. Exemple de folosire a *grep* pt. a căuta șiruri simple (fără regexp-uri):

```
grep xxx          ; nu s-a specificat input → așteaptă input de la user
xxx               ; xxx  (repetă căci a găsit șirul)
xxxy              ;  xxxy (repetă căci conține șirul
xyz               ; nimic – nu conține
CTRL-C            ; ieșire din modul interactiv
```

Simbolul | se citește ‘pipe’ (engl.) = redirectarea ieșirii unui program în intrarea următorului program:

```
echo xxx | grep xxxy       ; afișează xxxy; echo are efect de print în bash
echo xxx | grep xyz        ; nimic

cat >> fisier              ; creăm un fișier
un text
alt text
texte
CTRL-D                     ; salvează fișier

grep xxx fisier            ; caută în fișier șirul xxx, nu găsește
grep text fisier           ; printează toate liniile unde găsește
grep alt fisier            ; idem
```

Exemple de *regexp* (vezi o listă completă în [4])

.	orice caracter cu excepţia \n (*newline*)
A,a,1	orice caracter se potriveşte cu sine însuşi, cu excepţia caracterelor speciale . ^ $ * + ? { [ ] \ | ( )
\	pus în faţa unui caracter special pentru a-l face “nespecial”; de ex: \$ înseamnă caracterul $
[Aa1]	orice caracter din set (oricare din A *sau* a *sau* 1)
[1-3]	orice caracter din domeniul respectiv; aici, orice cifră între 1 şi 3
[a-zA-z]	orice literă mică sau mare
\d	orice cifră [0-9]
\w	echivalent cu [a-zA-Z0-9_] (*word characters)*
\W	orice caracter care nu e \w
\t,\n,\r	tab, newline, return
\s	[\n\r\t\f] (orice caracter de tip *whitespace*)
\S	orice caracter care *nu* este *whitespace*
^	la începutul şirului
$	la sfîrşitul şirului
+	1 sau mai multe apariţii a şirului care precede ‘+’; de ex: x+ şirul x *sau* xxx *sau* xxxx ...
*	0 sau mai multe apariţii a şirului care precede ‘*’; de ex: x* şirul gol *sau* x *sau* xxx *sau* xxxx ...
?	0 sau 1 apariţii a şirului care precede ‘?’; de ex: x* şirul gol *sau* x

```
grep text fișier      ; toate rîndurile conțin "text"
grep ^text fisier     ; doar ultimul rind conține subșirul "text" la început
grep text$ fisier     ; rîndurile care conțin "text" la sfîrșit
```

Folosind comanda linux 'ps' (*process status*), listați procese care să satisfacă anumite condiții;

```
ps                  listează procesele userului curent
ps aux              listează toate procesele, inclusiv cele din
                    background
ps aux | grep sshd  daemonul de ssh
ps aux | grep root  procesele care au cuvîntul 'root' undeva în cadrul
                    lor; după cum se vede nu-s doar procese care aparțin
                    de root
ps aux | grep ^root   acum doar procesele care aparțin de root, căci
                    trebuie să fie la începutul rîndului
```

Folosim *grep* pentru a opri (kill) un process care folosește prea multe resurse:

```
yes abcdefghijklmn          ; afișează în buclă șirul dat

CTRL-Z                      ; suspendăm procesul în background, nu-l oprim

ps aux | grep yes           ; verificăm că încă rulează

ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%cpu | head     ; ce resurse consumă?
```

îi aflăm PID-ul (Process ID, identificatorul unic de process) în coloana PID; aici este 9911:

```
 PID  PPID CMD                  %MEM %CPU
9802  2 [kworker/0:2]           0.0  1.4
9911  8476 yes abcdefghijklmn   0.0  0.6                ; vedem că e în top la %CPU
9978  1686 sshd: [accepted]     0.2  0.2
8354  1686 sshd: root@pts/0     0.4  0.1
1456  1 /usr/bin/vmtoolsd       0.2  0.0

kill -9 9911                ; -9 = terminare proces (echivalentă cu CTRL-C)

ps aux | grep yes           ; vedem că nu mai rulează
```

2. *Regular expressions în python*

În Python se importă modulul *re* pentru a avea acces la căutarea cu *regexp*

```
import re
match = re.search(pat, str)
```

metoda *re.search* caută patternul *pat* de tip *regexp* în şirul *str* şi în caz de succes întoarce rezultatul găsit în *match*. În caz de insucces se întoarce *None*

Exemplu: \d este un *regexp* care se potriveşte oricărei cifre 0-9; \d\d\d corespunde deci unui număr de 3 cifre. Întrucît caracterul "\" are semnificaţie specială în şiruri (vezi lucrarea introductivă), pentru a-l putea folosi şirul trebuie prefixat cu ***r*** de la *raw*, în care caz caracterul "\" nu se interpretează, ci se ia ca atare – permite șiruri în care există "\".

Rulaţi de mai multe ori programul de mai jos, modificînd şirul *str* să conţină, respectiv să nu conţină un număr format din minim 3 cifre, gen 123.

```
import re

str = 'sir care contine 123'
m = re.search(r'\d\d\d', str)
if m:
    print('Gasit: ', m.group())
else:
    print('Negasit')
```

Exemple (în comentariul după # este valoarea întoarsă; atenţie la "**r**" (*raw*) care precede şirul de *regexp* (nu este strict necesar decît dacă se foloseşte caracterul "\" în cadrul şirului, dar este recomandat pentru obişnuirea cu folosirea sa în orice *regexp*):

```
m = re.search(r'[a-c]' , 'a23')
print(m.group())                    # a

m = re.search(r'^123' , '1234')
print(m.group())                    # 123

m = re.search(r'^123' , '01234')
print(m.group())                    # None

m = re.search(r'\d\s*\d\s*\d', 'xx123xx')
print(m.group())                    # 123

m = re.search(r'\d\s*\d\s*\d', 'xx12  3xx')
print(m.group())                    # 12  3

m = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx')
print(m.group())                    # 1 2   3
```

Observaţie: în cazul în care sînt mai multe variante de şir care se potrivesc, regula de căutare este "*Leftmost and Longest"* – primul (de la stînga) şi de lungimea maximă pe acea poziţie:

```
m = re.search(r'i+', 'xiiziiii')
print(m.group())                    # ii    (Leftmost and Longest)
```

APLICAŢIA A1*

Scrieţi un program Python folosind *regexp*-uri care să primească de la user un şir de forma *'adresa mea de e-mail este ion@yahoo.com si-mi puteti scrie oricind'* (nu contează cuvintele exacte ci existența unei adrese de e-mail) şi să afişeze '*salut ion, văd că aparții de domeniul yahoo.com'*.

Indicaţie: în porţiunea de nume a adresei pot exista litere, cifre, precum şi semnele '.' şi '-'.

3. Dicţionare (*dicts)*

Un dicţionar e format din perechi *cheie:valoare* scrise între acolade {}. Se poate folosi pentru orice formă de asociere a unei categorii de date cu altă categorie, cum ar fi *hash tables*, perechi nume-adrese, etc:

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
pythonb.org	18.21.3.144
sourceforge.net	51.98.23.53

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Fig. 1 exemple de utilizare a dicţionarelor

```
dict = {}               # se porneste cu dict gol
dict['a'] = 'alpha'     # se introduc perechi pe rînd
dict['g'] = 'gamma'     # în ordinea dict[cheie]=valoare
dict['o'] = 'omega'

print(dict)             # {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}

print(dict['a'])     #  'alpha'
dict['a'] = 6        # schimbare
'a' in dict          # True
print(dict['z'])     #  KeyError
if 'z' in dict: print(dict['z']) # eliminare KeyError
print(dict.get('z'))             # None (in loc de KeyError)
```

O buclă *for* într-un dicţionar iterează implicit în cadrul *cheilor* (care vor apărea în ordine arbitrară). Metodele *dict.keys()* şi *dict.values()* întorc *liste* de chei şi valori. Metoda *dict.items()* întoarce o listă de *tupluri* (cheie, valoare). Listele pot fi sortate cu funcţia *sorted()*.

```
for key in dict: print(key)           #  a g o
for key in dict.keys(): print(key)    # la fel

print(dict.keys())    # ['a', 'o', 'g'] - este o listă
print(dict.values())  # ['alpha', 'omega', 'gamma'] - tot listă

for key in sorted(dict.keys()):     # iterare printre chei
  print(key, dict[key])

print(dict.items())  #[('a', 'alpha'),('o', 'omega'),('g', 'gamma')]

# această sintaxă for iterează în cadrul listei de
# tupluri .items() accesînd cîte un tuplu (key, value)
# la fiecare iteraţie

for k, v in dict.items(): print(k, '>', v)
# a > alpha    o > omega     g > gamma
```

Formatarea unui dicţionar folosind %: se inserează valorile dintr-un dicţionar într-un şir:

```
hash = {}                    # creare dicționar gol
hash['word'] = 'garfield'
hash['count'] = 42

# %d este int, %s este sir cu sintaxa de mai jos
s = 'I want %(count)d copies of %(word)s' % hash

s                 # rezultat: 'I want 42 copies of garfield'
```

APLICAŢIA A2*

Scrieţi un program python (folosind editorul) care:

- să citească un şir de la utilizator

- să creeze un dicţionar avînd drept chei literele din şir, şi drept valori numărul de apariţii a literei respective

- să afişeze aceste informaţii sub forma *litera: x frecvența: N*

Aplicațiile marcate cu * se vor verifica în vederea notării.

Bibliografie

[1] Python 3.x online docs: *https://docs.python.org/3/*
[2] Google Developers – Python course: *https://developers.google.com/edu/python/*
[3] Python exceptions: *https://docs.python.org/3/library/exceptions.html*
[4] Python regular expressions: *https://docs.python.org/3/library/re.html*