Programozzunk Pythonban (2. rész)
A szolid óriáskígyó

Műveletek akár a matematikában

Használtunk egyszerű műveleteket korábbi példáink során is, nézzük, mit tartogat még számunkra a ***Python***. Az osztással már találkoztunk, pontosítsuk kissé a már megismert adattípusok segítségével tudásunkat:

/ – az osztás eredménye, egész számot ad, ha egész számokat kap bemenő értékként, egyébként lebegőpontos eredményt kapunk
% – az osztás maradéka, ugyanolyan logikával dolgozik, mint az előző
// – az osztás egész számú eredménye, lebegőpontos számok osztásakor is mindig egészekre kerekít
Azaz például:

```
a=10.2
b=5
print('a%b='),a%b      #eredménye
                       #0.2
print('a/b='),a/b      #eredménye
                       #2.04
print('a//b='),a//b    #eredménye
                       #2.0
```

A számok világánál maradva megemlíthetjük még a hatványozást, melyet a **** jellel tudunk használatba venni, azaz például `13**2` jelenti 13 négyzetét.
A korábbi, mindennapokban is általánosan használt matematikai műveleteken túl a ***Python*** természetesen támogatja az informatika világára jellemzőbbeket is. Úgynevezett bitszintű művelet a >> és a <<, előbbi jobbra tolja el a biteket, utóbbi balra. Megértéséhez a kettes számrendszer alapjait kell kissé megismernünk, ami nem olyan bonyolult, mint hinnénk, csak egy csepp gondolkodást igényel.
A helyiértékek fogalmával már biztosan találkoztunk tízes számrendszerben, ebből induljunk most ki, vegyük például a 169-es számot:
169 jelentése: 1*100 + 6*10 + 9*1
másképp megfogalmazva $1*10^2 + 6*10^1 + 9*10^0$

Szemléletesen:

számjegy:	1	6	9
helyiérték:	10^2	10^1	10^0

azaz a tízes számrendszerben a helyiértékek 10 megfelelő hatványait jelentik, jobbról balra haladva folyamatosan növekedve (mindig 10^0 az első jobbról, majd balra haladva 10^1 következik, utána 10^2 , 10^3 stb.), s ezeket szorozzuk meg a számrendszerben érvényes számjegyek (0-9) egyikével.
Kettes számrendszerben is gondolkodhatunk ugyanígy, azaz kettő megfelelő hatványai következnek jobbról balra, s ezeket kell szoroznunk a számrendszerben érvényes számjegyekkel, melyek jelen esetben 0 vagy 1 lehetnek.
Jobbról balra haladva a helyiértékek így alakulnak, segítségképp a legalsó sorban kiszámolva őket:

számjegy:	[0]	[1]	[0]	[1]	[0]	[1]	[0]	[0]	[1]
helyiérték:	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
helyiérték:	256	128	64	32	16	8	4	2	1

Ha egy tízes számrendszerbeli számot kell átváltanunk, akkor azt kell megnéznünk, melyik az a legnagyobb szám ezek közül, amely még osztható vele – annak a helyiértékén 1-es számjegy fog szerepelni –, majd az osztás maradékát újra el kell osztanunk kettő hatványai közül azon legnagyobbal, mely még éppen osztója a maradéknak – az ennek megfelelő helyiértékre is 1-et írunk –, s az osztás maradékával ugyanezt kell tennünk egészen addig, míg olyan osztót nem kapunk, mely maradék nélkül megvan az éppen aktuális számban. Nézzük lépésről lépésre mindezt a 169 esetén:
Melyik az a legnagyobb kettő hatvány, ami még osztja? 128. Tehát a 128-nak megfelelő pozíciójú helyiértékre kell 1-est írnunk, azaz jobbról a 8. számjegy már megvan. Az előtte lévő pozíciókon biztosan 0 szerepel (körülbelül úgy, mintha tízes számrendszerben azt írnánk 0000000000000128, ami ugyanaz mint 00128 és természetesen egyenlő 128-al). Ezekre a bevezető nullákra csak akkor van szükség, ha például kifejezetten 16 számjeggyel akarjuk ábrázolni a számot, kettes számrendszerben ugyanennyi számjegy esetén ezt úgy hívják, hogy 16 biten ábrázoljuk. A bevezető nullákat kiírva, és a még ismeretlen helyiértéken lévő számjegyeket x-el jelölve ennyi biztos már az ábrázolásból:
0 0 0 0 0 0 0 0 1 x x x x x x x
Előző osztásunk maradéka 41, így most ehhez keressük a legnagyobb még osztónak megfelelő kettő hatványt. Ez a szám a 32, ami egyben azt jelenti, hogy jobbról a hatodik helyiértéken is 1-es szerepel, s mivel az osztandók egyre kisebb számok, ezért az előző helyiérték és e között lévő számjegyek már biztosan nem jönnek szóba, tehát azok 0-t vesznek fel:
0 0 0 0 0 0 0 0 1 0 1 x x x x x
Az előző osztásból maradt 9, az ehhez legközelebbi helyiérték a 8 lesz, azaz jobbról a negyedik pozícióba is 1-et írunk (s 0 kerülne az előző pozíciójú számjegy és e közé – ha lenne ilyen):
0 0 0 0 0 0 0 0 1 0 1 0 1 x x x
1 kettő hatványos osztóját keresve feltűnik, hogy ez az 1 és ezúttal nincs maradék, ami módszerünk szerint az is jelenti, hogy végeztünk, azaz
0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1

állapotból tovább már nem léphetünk. Így a 169 kettes számrendszerbeli megfelelője 16 számjegyet felhasználva (16 biten ábrázolva) 0000000010101001. Ha nem ragaszkodnánk a 16bithez, akkor rövidebben (8 biten ábrázolva): 10101001.

Négyzetrácsos papír helyett monitor

Írjunk egy rövid programot ***Pythonban*** az előző munkamenet alapján, ami helyettünk a tízes számrendszerből átváltja az adott számot kettesbe:

```
szam=169
bitek_szama=8        #azaz hány
# biten ábrázoljuk a számot

for i in range(bitek_szama,0,
➥-1):
   bit=szam//2**(i-1)      #a
# kiírandó számjegyek balról
# jobbra haladva
   print bit,        #kiírjuk a
# kiszámolt értéket, majd
   szam=szam%2**(i-1)      #a
# maradékkal folytatjuk a
# ciklust
```

A programban van egy számunkra még ismeretlen vezérlési szerkezet, mely lehetővé teszi, hogy addig ismétlődjön a számjegybitek kiszámítása, amíg ki nem írtunk a `bitek_szama` változóban beállított mennyiségű számjegyet. Valójában nem is egy, hanem két egymásba ágyazott szerkezetről van szó, melyből a `for` több programnyelvben általánosan használt ciklusvezérlő. A ***Python*** kissé más logika szerint használja mint például a ***C*** nyelv, inkább egy karakterlánc (string) típusú felsorolás, lista esetén tudnánk hatékonyan kihasználni képességeit, mivel előre megadott értékek mentén képes ciklust szervezni. Például a következőképp tudunk egy lista elemein végigmenni, és kiíratni azokat:

```
listank=[1, 'kutya', 'macska',
➥13, 'pingvin']
for ciklusvaltozo in listank:
   print listank
```

Ha pedig csak arra lenne szükségünk, hogy egytől százig írja ki a számokat az értelmező, akkor legegyszerűbben talán így kérhetnénk tőle:

```
range(100)
```

Gyakorlatilag csak a felső korlátot adtuk meg, a kezdőértéket (1) és a lépésközt (1) a beépített függvény „kitalálta”. Természetesen megadhatunk más záróértéket és lépésközt, ezt tettük a példaprogramban is, azaz a ciklusmag számol visszafelé (lépésköz -1) egészen 0-ig, a megadott `bitek_szama` változótól kezdve, s az éppen aktuális értékét az `i` változóban tároljuk.
A kettes számrendszer fent részletezett logikájából következik, hogy a 8 biten ábrázolható legnagyobb szám $2^{(8-1)}$, ezért láthatjuk azt, hogy a ciklusban már csak $2^{(i-1)}$. hatvánnyal osztunk.
A `print bit,` utasítás után szereplő vessző szándékos, segítségével a számjegyek nem sorban egymás alá (különben a `print` automatikusan sortöréssel dolgozna, mintha `\n` sorvégjelet kapott volna), hanem szóközzel elválasztva egymás mellé kerülnek a jobb átláthatóság érdekében. Igazi megoldást (szóköz nélkül, egy számként látszódva ábrázolni) a `print` helyett más függvény használata jelentene, erre a későbbiekben kitérünk.

Túlcsordulunk

Mi történik, ha kevesebb biten próbáljuk meg ábrázolni a számot, mint amennyit az igényelne? Ha például a `bitek_szama` változó értékét 4-re állítjuk, a következő eredményt kapjuk:
21 0 0 1

Mi változott az eddig jól működő programban? Semmi, pusztán elfelejtettünk felkészülni a megszokott normálistól eltérő szituációkra. A program futtathatóságát, azaz nyelvtani helyességét illetően nincs aggályunk (szintaktikailag helyes), viszont így már a logikája rosszul működik (szemantikailag hibás). Tegyünk bele egy egyszerű ellenőrző rutint:

```
szam=169
bitek_szama=8        #azaz hány
➥biten ábrázoljuk a számot

for i in range(bitek_szama,0,
➥-1):
   bit=szam//2**(i-1)      #a
# kiírandó számjegyek balról
# jobbra haladva
   if bit<=1:
      print bit,
   else
      print "Szerintem nem lehet
➥ennyi biten ábrázolni!"
      break       #kilépünk
# a ciklusból és a program
# véget ér
   szam=szam%2**(i-1)      #a
# maradékkal folytatjuk
# a ciklust
```

Az újonnan elhelyezett feltételvizsgálatban `break` utasítás megszakítja a ciklus működését, és kilép a programból ha kettes számrendszerbe nem illő számjegy keletkezne, viszont a feltételes vezérlés (`if`) használata miatt ha elegendő a megadott bitszám, akkor az utána következő művelet végrehajtását ez egyáltalán nem befolyásolja.
Ezek után könnyedén fel tudjuk írni például a 25-öt kettes számrendszerben 8 bittel ábrázolva:
00011001

A << műveleti jellel balra toljuk egy bittel az egész számsort, így binárisan 00110010-t kapunk:

```
print('25 egy bittel balra
➥forgatva:' ,25<<1)
50
```

Ahol eddig nem szerepelt számjegy, oda 0-t írunk (jobb szélső érték), ill. az egész számsort balra csúsztattuk el. Mindez ugyanazt eredményezi, mintha kettővel szoroztunk volna, hiszen a kapott szám éppen 50. Ha nem egy, hanem két helyiértékkel csúsztattuk volna balra (`50<<2`), akkor az megfelel a néggyel való szorzásnak (100 az eredmény), `50<<3` ugyanaz, mintha nyolccal szoroznánk stb., azaz a jel jobb oldalán szereplő számot kettő n-edik hatványkitevőjeként használva $50*2^n$ értékeket számoltathatunk ki e példában.
Ennek nagyjából ellenkezője történik a jobbra csúsztatás során, ha kiadjuk:

```
50 >> 1
```

az eredmény 25 lesz, mivel a bitek jobbra csúszva 00001100 -t adnak kettes számrendszerben, s az eredetileg jobbszélső helyiértéken szereplő 1-es csonkolásra került, más néven túlcsordult. Programunk esetén is azt tapasztalhattuk, hogy a szám nagysága és az ábrázolandó bitek száma közötti

összefüggés miatt előfordulhat, hogy nem tudjuk a számot kettes számrendszerbe korrekten átszámítani. Még látványosabb ugyanez a művelet, ha nem eggyel, hanem négy helyiértékkel csúsztatjuk jobbra az 50-et:

```
print '50 >> 4 eredménye: ',
➥50>>4
50>>4 eredménye: 3
```

Ugyanazt az eredményt kaptuk, mintha maradék nélkül osztunk 2^4 -el, vagyis kiadtuk volna:

```
50//16
```

Megemlíthetjük még a bitszintű tagadást, jele a ~, értékét pedig a következő egyenlőség mutatja:

```
~szám = -1*(szám-1)
```

Mindez a negatív számok tártakarékosabb, úgynevezett kettes komplemens ábrázolásával függ össze, s ezért nem egyszerűen ellenkezőjére váltja a biteket.
Segítségül hívhatjuk a ***Pythont*** akkor is, ha vissza akarunk váltani egy számot kettes számrendszerből tízesbe, de akár nyolcasból tízesbe is, mégpedig a már ismert `int()` konverziós függvény segítségével:

```
a=int('0110',2)      #a második
# argumentum jelzi, hogy hányas
# számrendszerből
print 'a értéke szerintünk
➥tízes számrendszerben 6,
➥a Python szerint:' , a
b=int('644',8)    #nyolcas
# számrendszerből alakít át
print 'b értéke szerintünk
➥tízes számrendszerben 420,
➥a Python szerint:',b
```

Figyeljük meg, hogy az `int()` hívásakor egyszeres vagy kétszeres idézőjeleket használtunk, különben a konverzió nem sikerül, mivel egyszerű karakterláncot, és nem „igazi számot" vár tőlünk az értelmező.

És a jogaink? Vagy a jogaink?

Hol láthatjuk még hasznát a kettes számrendszernek? A bitszintű operátorok használatakor például áramkörök működésének megértésekor, vagy akár gyorsítás céljából alacsony szintű programozás esetén, háromdimenziós grafikánál, s más alkalmazásokban.
Bitszintű ***és*** művelet jele ***&***, értéke két 1-es számjegy találkozásakor 1, minden más esetben 0.
Kettes számrendszerben felírva a két számot:

egyik szám:	1001 (tízes számrendszerben értéke 9)
másik szám:	0101 (tízes számrendszerben értéke 5)
eredmény:	0001 (tízes számrendszerben értéke 1)

Bitszintű ***vagy*** művelet jele `|` , jelentése pedig az, hogy azonos helyiértéken két 0 számjegy találkozása ad 0-t, minden más esetben 1 fog szerepelni.

egyik szám:	1001 (tízes számrendszerben értéke 9)
másik szám:	0101 (tízes számrendszerben értéke 5)
eredmény:	1101 (tízes számrendszerben értéke 13)

Mire használhatjuk ezt a tudást a gyakorlatban? Nézzünk egy egyszerűnek tűnő példát, a fájljogosultságokat.
Ha egy adott könyvtárban kilistázzuk az állományokat, a következőhöz hasonló kimenetet kapunk:

```
[valaki@localhost]$ ls -l
total 24
-rwxr-xr-x 1 valaki valaki 462
➥Oct 17 10:21 pelda.py
-rwxr-----  1 valaki mas
➥80 Oct 10 23:02 teszt.txt
```

Az első oszlopblokk reprezentálja a jogosultságokat, ***Unix/Linux*** felhasználók számára feltehetően ismerős séma szerint, azaz az első érték – vagy `d` lehet (utóbbi jelenti a könyvtárat), ezt a továbbiakban figyelmen kívül hagyjuk; a következő három a felhasználó hozzáférési jogosultsága, majd három karakter jelzi a csoportét, végül három a többiekét.
Számunkra lényeges, hogy e három különálló csoportot viszonylag egységesen lehet kezelni, hiszen ha kiadjuk a `chmod - x pelda.py` parancsot, akkor mindhárom csoportnak egyszerre szüntetjük meg az addig meglévő futtatási jogát az adott fájlra. A ***chmod*** parancs azonban ennél finomabban is szabályozható (csoportra, egyénre), legegyszerűbben talán úgy, ha a jogokat számokkal reprezentáljuk.
E számokat úgy kell elképzelnünk, mintha az ***rwx*** jelzők mindegyike egy-egy bitet jelentene, s a karakterhármas együttesen egy oktálisként (nyolcas számrendszer, gyakorlatilag mintha 3 egymás mellett álló bit képezne egy számot) mutatná a jogokat. Szemléletesebben az előző példánk a ***pelda.py*** fájlra, ahol 1 az adott bit értéke, ott van beállítva a jogosultság:

```
d r w x r w x r w x
0 1 1 1 1 0 1 1 0 1
```

Minden számhármas értelmezhető egy-egy háromjegyű bináris számként, melynek maximális értéke számhármasonként átszámítva 7 (111 binárisan), minimális értéke pedig 0 (000 binárisan), a köztes értékeket pedig aszerint veszik fel, hogy mely jogosultság van beállítva. Például a tulajdonosnak van futtatási, a csoportnak írási és olvasási joga, mindenki másnak csak írási joga, és nem könyvtárról van szó:

oktálisan:	0	1	6	2
binárisan:	000	001	110	010
leírva:	d	rwx	rwx	rwx

Tételezzük fel, hogy egy fájlunk hozzáférési jogosultságát kell beállítanunk úgy, hogy a tulajdonos számára írható legyen, de minden más attribútuma változatlan maradjon. Ezt úgynevezett maszkolással lehet könnyen megoldani (lásd ***umask*** működése), ahol egy olyan bitmaszkkal végzünk ***vagy*** műveletet, mely minden más bitje 0 értékű (így a ***vagy*** művelet működéséből következően nem változtatja meg az eredeti értékeket) a beállítani kívánt pedig 1.

```
#!/usr/bin/python
import os,stat,sys

fajlnev=sys.argv[1] #parancs-
# sori indítás paramétere
informacio=os.stat(fajlnev)[ST_
➥MODE]  #modulból stat futtatás
# jogosultságbitekre
print "Jogosultságok oktális
➥számmal:' , oct(jogok)   #ki-
# íratjuk oktális számokkal
```

Az `import` utasítással betöltjük a használni kívánt – a nyelv alapkészletét bővítő – modulokat, melyekkel a viszonylagos hordozhatóságot (`os`) és az operációs rendszer parancsaihoz hozzáférést (`stat, sys`) tudjuk biztosítani. Működésük pontosabb megértését segítheti a dokumentáció böngészése, de hatékony használatukhoz szükségünk lesz a később ismertetendő objektum-orientált tervezési elvek ismeretére is, egyelőre csak vegyük igénybe őket. Az `ST_MODE` segítségével a jogosultságbiteket és egyéb jellemzőket (***sticky bit, fájltípus*** stb.) tudjuk akár makrószerűen egyesével lekérdezni. Minket most csak a három csoport jogosultságai érdekelnek, ezért egyszerűsítjük a kiíratást:

```
eredeti_oktalis_jogok=oct(jogok
➥& 0777)
print "Könyvtár,tulajdonos,
➥csoport,egyéb oktálisan:' ,
➥eredeti_oktalis_jogok
```

Láthatjuk, hogy a megjelenített érték egyezik az előző, szűretlen kiíratás utolsó három számjegyével (a számkezdő 0 csak azt jelzi, hogy oktális szám, és nem tízes számrendszerbeli), hiszen ***bináris és*** műveletet hajtottunk végre egy csupa 1-esekből álló (777 oktálisan) számmaszkkal, ezáltal az eredeti bitek értékét nem változtattuk. Pontosabban csak a jobb oldali három bit értékét, az első négy bitet ugyanis pont ezzel – a bevezető nullákat is oda kell gondolni – nulláztuk le.

A `pelda.py` fájl `1 1 1 1 0 1 1 0 1` jogosultságbitjeit (lásd feljebb) oktálisra váltva 755-öt kapunk, ezt kapcsoljuk össze ***és*** műveleten keresztül a 777-es maszkkal, melynek utolsó három számjegye binárisan 111111111. Egyszerűsítve:

```
eredeti pelda.py fájl oktálisan:
0100755
maszk oktálisan:          0000777
-----------------------------
-----------------------------
eredmény oktálisan:       0000755
```

Ez a maszk egyelőre csak annyit csinált, hogy „levágta” a fölösleget, a számunkra jelenleg érdektelen bevezető bitekkel így nem kell foglalkoznunk. Ejtsünk néhány szót a `fajlnev=sys.argv[1]` sorról is. Eddigi példáinkban a változókat jobbára előre feltöltöttük értékkel, rugalmatlan programozási megoldást választva. Ha annak eldöntését, mely fájl jogosultságait írassuk ki s módosítsuk majd, a felhasználóra szeretnénk bízni, legegyszerűbb, ha parancssori argumentumként kérjük be. A gyakorlatban ez annyit tesz, hogy

pythonprogramnév argumentum1 argumentum2 argumentum3 … argumentumn

alakban kell elindítani a programot, s mi a `sys` modul segítségével ezen argumentumok értékeit egy számozott listából ki tudjuk olvasni.
A legelső argumentum maga a futtatott fájl – függetlenül attól, hogy megadjuk-e külön, erre `argv[0]`-ként lehet hivatkozni, míg a többi sorban a következő sorszámú elem, azaz az első „igazi” parancssori argumentum az `argv[1]` lesz.
Esetünkben így lehet a szkriptet a ***pelda.py*** fájlra értelmezve futtatni:

```
pythonprogramneve pelda.py
```

Folytassuk rövid programunkat a lekérdezés után a jogosultságbeállítással. A használandó maszknak csakis a tulajdonos írási bitjét szabad 1-re változtatnia (ez jobbról a 8. bit), az összes többi bitet változatlanul kell hagynia, akármilyen értékük is volt eredetileg. Erre alkalmas a ***vagy*** művelet és egy ilyen bináris bitmaszk: 0 1 0 0 0 0 0 0 0, ami oktálisan kifejezve 200.

```
maszk=oct(0200)   #tulaj_rwx=2
# csoport_rwx=0 mások_rwx=0
beallitott=int(eredeti_oktalis_
➥jogok,8)  | int(maszk,8)
# tízes számrendszerbe váltva
# VAGY
os.chmod(fajlnev,oct(beallitott
➥& 0777)   #a művelet eredménye
# szerint állítjuk be
```

Amint látjuk, a biztonság kedvéért átváltottuk egyazon számrendszerbe a műveletek előtt az oktálisokat, majd átadtuk a kapott értéket az operációs rendszer használó modulnak.
A ***chmod*** két paramétere értelemszerűen a fájlnévből és az új jogosultságból áll, ami „tartalmazza” a korábbit, és gyakorlatilag egyenértékű a `chmod u+w` parancs kiadásával.

A	B	A ÉS (&) B	A VAGY(|) B	A KIZÁRÓVAGY(^) B
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

1. ábra Igazságtábla

Az előző ***vagy*** műveletet ***megengedő vagy***ként is emlegetik, ellentétben a ^ jellel jelölt ***kizáró vagy***-gyal, mely a számjegyek találkozásakor csak akkor ad 1-est eredményül, ha két különbözőről van szó, azaz 0 és 1 vagy 1 és 0 ugyanazon a pozíción:

egyik szám:	1001 (tízes számrendszerben értéke 9)
másik szám:	0101 (tízes számrendszerben értéke 5)
eredmény:	1100 (tízes számrendszerben értéke 12)

Végezetül használjuk fel ezen tudásunkat egy egyszerű titkosítóprogram írására, melyhez számokat és a ***kizáró vagy*** műveletet hívjuk segítségül.

```
adat=17
kulcs=35
titkositva=adat^kulcs
print"Az adat titkosítva így
➥néz ki:' , titkositva  #ez
# azért nem túl biztonságos
# titkosítás...
print "Visszafejtése:
➥titkosított adat kizáró_vagy_
➥művelet kulcs, azaz:"
print titkositva^kulcs
```

Alacsony szintű, bináris műveletekkel ritkábban találkozunk közvetlenül, mint például a későbbiekben ismertetett logikai és,vagy vagy egyéb kifejezésekkel. Nem árt tudnunk azonban, hogy egy magas szintű utasítás végrehajtása mögött milyen elemi lépések lehetnek, illetve bizonyos helyzetekben kifejezetten jól jön ez a tudás sebességnövelésre, egyszerűsítésre. A ***Python*** szerencsére egyre segítőkészebb ahogy haladunk vele előre, s a magasabb szintű funkciók esetében könnyebb dolgunk lesz.

Tóth Virgil Zoltán
(m_v@c2.hu)