Programozzunk Pythonban (3. rész)

Feltételes szerkezetek, listák, operátorok

Logika ismét, magasabb szinten

Programozással megoldható feladatokban gyakran „csak” egy kijelentés igazságtartalmát szeretnénk megtudni, s nem érezzük szükségét annak, vagy nem látjuk értelmét, hogy a biteket vizsgáljuk. Ilyenkor jól jönnek a magasabb szintű logikai műveletek. Sok más programozási nyelvhez hasonlóan a Python is azt az elvet követi, hogy a 0 értéket tekinti hamisnak, és minden mást igaznak vesz. Természetesen ehhez előbb saját magának elvonatkoztatva 0-ra és – általában az egyszerűség kedvéért – 1-re kell alakítania a kiértékelni kívánt kifejezést. Például nézzük meg újra a számok párosságát vizsgáló rövid kis programot átírva:

```
a=15
if a%2:    #mivel 15-öt 2-vel
#osztva a maradék 1,
#ez a kifejezés IGAZ
  print "Páratlan"
else :
  print "Páros"
```

Az eredeti példához képest kénytelenek voltunk megfordítani a vizsgálat eredményét kiíró részeket, így viszont megspórolhattuk az összehasonlítást (emlékezzünk rá, hogy eredetileg `if a%2 == 0:` szerepelt). A kiértékelés során az `if a%2:` sort az értelmező igaznak tekinti (mivel a maradék 1, azaz 0-tól különböző), ezért esetünkben a vezérlés az első szöveg kiírásával ér véget. Talán kevésbé nyilvánvaló a példa, ha nem számokat, hanem szöveges tartalmat kell elemezni:

```
elso=''  #üres sztring, két '
#jel között nincs semmi
masodik='Nem vagyok üres'
if elso:
  print "elso: IGAZ"
elif masodik:
  print "elso: HAMIS, masodik:
  ➥IGAZ"
else:
  print "elso: HAMIS, masodik:
  ➥IGAZ"
```

A feltételvizsgálat kissé bonyolultabbnak tűnhet, mint az eddigiek, valójában csak az összevetés további finomításáról van szó. Ha az első feltételnek (`if`) nem felel meg a kifejezés, akkor hajtódik végre az `elif` ág, s ha ebbe a kategóriába sem tartozik, akkor a végső `else` ág. Természetesen az `elif` ágakból lehetne több is, működésük hasonló más programozási nyelvek ***case – switch*** párosához. A példa értelemszerűen azt fogja kiírni, hogy az első vizsgálat hamis eredményt, míg a második igazat hozott, mivel egy üres karakterláncot nézve az értelmező automatikusan 0 logikai értékre alakította át saját maga számára. Lehetőségünk van a már megismert bináris logikai műveletek magasabb szintű (úgynevezett ***Boole algebra*** szerinti) használatára is. Ha két kifejezés közé egy ***and*** műveletet illesztünk, az előző példát egyszerűbben tudjuk leírni:

```
elso=''  #üres sztring, két '
#jel között nincs semmi
masodik='Nem vagyok üres'
if elso and masodik:
  print "IGAZ mindkettő."
else:
  print "HAMIS valamelyik
  ➥feltétel."
```

Egy nagy különbség mindjárt szembeötlik: utóbbi módszerrel csak azt tudtuk megmondani, hogy valamely feltétel hamis volt, azt nem, hogy melyik. Az ***és*** logikai művelet végrehajtása során az értelmező előbb megvizsgálja az első feltételt, s ha azt hamisnak találja, már meg sem nézi a többit, automatikusan hamis értékűnek tekinti az egész kifejezést. Ha az első igaz, a másiktól függ a kimenet, annak igaz volta az egész kifejezést igazzá teszi, hamis érték esetén pedig az egész hamis lesz.
Az `or` nevezetű ***vagy*** művelet működése megegyezik a bináris ***megengedő vagy*** műveletével, azaz csak akkor ad hamis értéket, ha mindkét feltétele hamis, minden egyéb esetben igaz lesz a végeredmény:

```
elso=''  #üres sztring, két '
#jel között nincs semmi
masodik='Nem vagyok üres'
harmadik='' #üres sztring, két '
#jel között nincs semmi
if (elso or masodik) and
➥(elso or harmadik):
  print "Az elso VAGY masodik,
  ➥valamint az elso VAGY
  ➥harmadik művelet
  ➥végeredménye IGAZ"
elif elso or harmadik:
  print "Az elso VAGY harmadik
  ➥művelet végeredménye IGAZ"
else:
  print "Az elso VAGY harmadik
  ➥művelet végeredménye
  ➥biztosan HAMIS, a többit
  ➥nem tudom"
```

A kiértékelés során az első feltétel igaz ugyan (`elso or masodik`), de a feltételvizsgálatba ezúttal két logikai kifejezés együttes vizsgálatát tettük, s a második feltétel (`elso or harmadik`) már hamis értéket eredményez. Ezért a vezérlés a következő ágon

folytatódik,s mivel ott már konkrétan csak az `elso` és a `harmadik` nevezetű változókat kapcsoltuk össze vagy művelettel, ennek eredménye egyértelművé teszi, hogy mindkettő hamis-e (igen, tehát az utolsó sort írja ki a program). Ettől még az első feltételvizsgálat eredményéről nem tudtunk meg többet, azaz az `elso` és `masodik` nevű változó viszonya ebből a programból nem derül ki. Szándékosan hoztam ilyen látszólag esetlen példát, talán ebből jól látható, hogy az egyszerű logikai műveletek során is nagyon figyelnünk kell, mi kerül egyáltalán a vizsgálat látóterébe, különben az információk hiányában rosszul következtethetünk.
A logikai tagadást a `not` szócska jelenti, az igazból hamis, a hamisból igaz értéket képez:

```
a = 0
if not a:
  print "Így már nem nulla
  ➥logikailag, azaz IGAZ
  ➥értéke viszont maradt:" , a
else:
  print "Logikailag is nulla,
  ➥azaz HAMIS értéke pedig:"
  ➥, a
```

Szót kell ejtenünk még a ***nem egyenlő*** jelentéssel bíró `!=` operátorról is, mely a már többször használt == egyenlőségvizsgálat ellentettje.

```
a = 0
b = '0'
if a != b:
  print "Nem egyenlő a(" , a ,
  ➥") és b(" , b , ") értéke"
else:
  print "A két érték egyenlő"
```

Túlterhelt operátorok

A korábbi példákból kitűnik, hogy bár a ***Python*** a szkriptnyelvek többségéhez hasonlóan nem tartozik a szigorúan típusos nyelvek közé, néhány alapvető adattípust meg kell különböztetnünk a hatékony használat érdekében. Van ugyan átjárhatóság a számok és a karakteres adatok között (erre volt jó példa a számnak látszó karaktersorozat felhasználása összeadásban), de ügyelni kell a megfeleltetésekre.
Említettük az általános jellemzésben, hogy a ***Python*** támogatja az objektum-orientált programozást és tervezést, sőt, e szemlélet egyik legfontosabb tulajdonsága e nyelvnek. Lehetséges például az operátor túlterhelés néven ismert jelenség felhasználása programjainkban. Ez alatt azt értjük, hogy egy eredetileg valamilyen célt szolgáló operátort (példánkban a számok összeadását jelképező + jel) teljesen más tulajdonsággal ruházunk fel, jelen esetben karakterláncokat kapcsolunk össze segítségével.
Íme egy példa:

```
a=14
b=27
c='császár'
d='pingvin'
print "Az a változó értéke: ",
➥a, " típusa: " , type(a)
print " A b változó értéke: ",
➥b, "típusa: ", type(b)
print "Az a + b művelet
➥értéke:", a+b, " tehát a +
➥operátor itt összeadást
➥jelentett."
print "A c változó értéke:", c,
➥" típusa: ", type(c)
print "A d változó értéke:", d,
➥" típusa:", type(d)
print "A c + d művelet
➥értéke:", c+d, " tehát a +
➥operátor itt összefűzést
➥jelentett."
```

Talán kevésbé meglepő ezek után a következő példaprogram kimenete:

```
a='császár'
b='pingvin'
c='ek'
if a+b+c != (a+b)+c:
  print "E két művelet
  ➥eredménye nem egyezik"
else:
  print "A " , a+b+c , " a " ,
  ➥a+b ,  " többesszáma, \n\
    csakúgy, mint a " , (a+b)+c
print "Három ilyen állat: " ,
➥3*(a+b) , "\n\
  nem ugyanaz, mint  három
  ➥ilyen állat: " , 3*a+3*b
```

Mit is láthattunk itt? Az operátortúlterhelés eredményeként összefűztünk három sztringváltozót, és az összeadás eredeti, asszocivitás néven ismert csoportosíthatósági tulajdonságával találkoztunk. Ráadásul a számok esetén szorzást jelentő operátort is túlterheltük, és segítségével a karaktersorozatokat sokszoroztuk meg. A szorzásjellel jelölt művelet végrehajtása ugyanúgy korábban történik, mint az összeadásjellel jelölteké (erre a precedenciasorrendre példa az utolsó sor). A karaktersorozatok jellegéből fakadóan azonban nem kommutatív a sokszorozás az összefűzésre nézve, azaz az utolsó két sor eredménye eltérő, noha az eredeti, számokkal végzett művelet esetén nem lenne különbség. (Az összefűzés művelete már önmagában sem felcserélhető, hiszen a végeredményben található karakterek megegyeznének ugyan, de értelmük teljesen más lesz.)
Az operátortúlterheléstől függetlenül újdonságot jelent a több sorba tördelt összetartozó szöveg is, erre lehetőséget a példaprogramban használt backslash biztosít, míg az előtte álló `\n` karakterek azt jelzik, hogy sortörést is kérünk. Ügyeljünk rá, hogy a lezáró ” jelet a következő sorban kell az adott szövegegység végén elhelyezni, különben az értelmező hibát jelez.

Listák, tuple

Az eddig részletezett egyszerű adattípusokon túl lehetőségünk nyílik összetett adattípusokat is igénybe venni. Ezek közül a listával már találkoztunk egy példa erejéig, új alkotása egyszerű:

```
lista=['13' , 43.7 , 1988 ,
➥'kutya' , 'macska']
```

A lista úgynevezett szekvenciális adattípus, sorban haladva érhetők el elemei. Hivatkozni rájuk sorszámuk alapján lehet, 0-tól kezdve a számozást, a lista neve után egy szögletes zárójelbe foglalva Jellemzője továbbá, hogy bővíthető illetve szűkíthető tetszőlegesen, és elemeinek nem kell egyforma típusúaknak lenniük. Például:

```
honap='január'
hideg_honapok=['november'  ,
➥'január' , 'február' ]
if honap in hideg_honapok:
  print honap , "is a hideg
  ➥hónapok között szerepel, \n\
     csakúgy, mint például " ,
     ➥hideg_honapok[0]
```

Ciklusszervezést már láthattunk, az itt használt hasonlít a `for i in range()` típusúakra, viszont itt a feltételt nem a hagyományos értelemben vett ciklusváltozóval állítjuk be, hanem egy tartalmazásra kérdezünk rá. Ha csak végig akarunk menni a lista elemein, a következő ciklust érdemes használnunk:

```
hideg_honapok=['november'  ,
➥'január' , 'február' ]
print "A hideg_honapok lista" ,
➥len(hideg_honapok) , " elemet
➥tartalmaz:"
for elemek in hideg_honapok:
  print elemek
```

Könnyű dolgunk volt, nem kellett foglalkozni a lista kezdő- és végpontjával, a ciklus minden lényeges teendőt ellátott helyettünk. Érdekességképp kiírattuk a lista hosszát a `len()` függvény segítségével, mely nem csak a listák, de egyéb adatszerkezetek esetén is hasznos. Ha saját magunknak kellene algoritmust gyártani egy lista bejárására, körülbelül így nézne ki:

```
lista = ['elem1' , elem2,
➥'elem3', … elemn]
cilusváltozó = 0
listavége = elemszám(lista)
ciklus ciklusváltozó
➥kezdőpontjától amíg
➥ciklusváltozó <  listavége:
  tároló változó = lista
  ➥[változó. eleme]
  kiíratjuk tároló változót
  cilusváltozó =
  ➥ciklusváltozó+1
ciklus vége
```

Próbálkozzunk meg néhány egyszerűbb listamanipulációval:

```
honapok=['november' , 'január' ,
➥'február' ]
honapok[1] = 'december'  #a
#második elem helyére tesszük
```

Konkrét helyet jelöltünk ki az elem számára, és mivel ott már szerepelt egy érték, ezt felülírtuk.

```
utolso_elem=honapok.pop()     #a
#lista legnagyobb indexű eleme
print utolso_elem
```

A lista utolsó elemét a `pop()` függvény segítségével olvastuk ki. Az objektum-orientált szemléletnek köszönhetően a ***Pythonban*** szinte minden – így a lista is – objektum. A használt függvény ennek egy metódusa, azaz hozzá tartozó függvénye, – ezt jelzi a lista neve után tett pont – melynek segítségével a külvilággal kommunikál. A memóriaszervezésben, tárfoglalásban van elsődlegesen szerepe az utoljára elhelyezett adatoknak, ezért is hoztak létre külön függvényt számára. Több olyan metódust is találhatunk, melyek a listákkal való hatékony munkavégzést segítik, ezek felkutatásában segíthet például a parancssorból kiadott `pydoc list` parancs.

```
del honapok[0]    #töröljük
#novembert
mas_honapok = ['március' ,
➥'április' , 'május' ,
➥'június']
honapok.extend(mas_honapok)
   #hozzáadunk egy teljes
#listát
honapok.append('12')   #egy
#elemet adunk hozzá
```

Jól látható a különbség egy elem hozzáadása és egy teljes lista (vagy egyéb szekvenciális adat) hozzáadása esetén: más függvényt használtunk. Törölni már az előzőektől eltérően egy külön utasítással tudtunk, azonban lehetőség van a listához tartozó `remove()` függvény használatára is, de figyeljünk a különböző lehetőségek különböző zárójeleire (`[]` között az elem sorszáma, `()` között a neve):

```
honapok.remove('december')
```

Arra is lehetőségünk van, hogy egyszerre hozzáadjunk valamit az összes elemhez, például írjuk mögéjük a „hónap” szót:

```
honapok = [elemek+' hónap' for
➥elemek in honapok] #bővítés
#minden elemnél
for kiir in honapok:
  print kiir
```

Az eddig használt megoldásoknál életszerűbb a tartományra hivatkozás, melyet a tartományokra vágás műveletével tudunk elérni:

```
honapok[2:]=['új elem']  #a 3.
#elemtől töröl, és ezt írja
#bele egyszer
del honapok[1:3]   #az 1. és a
#2. elemet törli
del honapok[:3]    #a 3. elemig
#töröl, csak a 4.-től maradnak
#meg
```

Komoly odafigyelést igényel eleinte a szeleteléses technika elsajátítása, hiszen figyelni kell a 0-val kezdődő indexszámokra, valamint arra is, hogy a szeletelés által megadott számok nem az elemeket vagy az indexet, hanem az indexszám előtti pozíciót jelölik.

Például:

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del lista[3:6]'
print lista

A végeredmény [1, 2, 3, 7, 8, 9] lesz, azaz a harmadik indexszámú (4) elem bal oldalától a hatodik indexszámú elem (7) bal oldaláig vágtunk ki a listából.
Ha csak annyit tudunk, hogy egy adott elem szerepel a listában, és például vele együtt a harmadik elemig bezárólag szeretnénk kivágni egy darabot, akkor segíthet az `index()` függvény:

```
del honapok[(honapok.index
➥('március hónap')):3]
```

A listákhoz nagyon hasonló adattípus az úgynevezett ***tuple***, két jelentős különbséget érdemes azonban megemlítenünk: nem változtathatóak az elemei, és nem szögletes zárójelbe kell őket tenni:

```
honapok_tuple = ('január',
➥'február', 'március')
```

Szekvenciális és egyéb adatszerkezetek segítségével már kisebb programokat érdemes írni, melyek nagymértékben megkönnyíthetik az adatfeldolgozáshoz, szövegmanipulációhoz kapcsolódó feladatokat. S mindeközben észrevétlenül ismét közelebb jutottunk az objektum-orientált szemlélethez is.

Tóth Virgil Zoltán
(m_v@c2.hu)