# Programozzunk Pythonban (4. rész)
# Függvények

Szekvenciális adattípusokkal foglalkoztunk legutóbb, azonban óhatatlanul találkoztunk az adatmanipulálásra alkalmas, ismétlődő szerkezetekkel, azaz a függvényekkel is. Ezek közös jellemzője, hogy gyakran használt, vagy logikailag jól tagolható műveleteket fognak össze egy jól definiált csoportba, s nevük meghívásával és a szükséges paraméterek átadásával tudjuk őket hasznosítani.

A függvények egy speciális fajtájaként tekinthetünk az úgynevezett ***metódusok***ra vagy tagfüggvényekre, ezek az objektum orientált fejlesztés kellékei. Egyszerűen fogalmazva az objektumok olyan egységbe zárt információhordozók, melyek a program többi részével vagy külső alkalmazásokkal egy jól ellenőrizhető csatornán keresztül kommunikálnak csak, azaz a metódusok segítségével lehet hozzáférni a bennük tárolt adatszerkezetekhez. Ennek a megközelítésnek legfőbb erénye a védhetőség (véletlen vagy szándékos adat- és szerkezeti módosítások ellen) és a méretezhetőség, valamint a könnyű használat (nem szükséges ismernünk az objektum belső szerkezetét ahhoz, hogy használni tudjuk, elég csak a metódusokhoz hozzáférni). Ily módon már többször is igénybe vettük a ***Python*** alaprendszer nyújtotta szolgáltatásokat, gondoljunk csak a listamanipuláló utasításokra. Természetesen az objektum előbbi definíciója még jócskán pontosításra és bővebb kifejtésre szorul, erre azonban később kerítünk sort.
Beépített függvény volt például az egyik numerikus adattípus konverziójáért felelős `int()` :

```
x = '13'
szam = int(x)
```

A függvény neve után kerek zárójelben a feldolgozásra átadott paraméterek állhatnak, de arra is akad példa, hogy nem adunk át paramétert:

```
help() #interaktív segítség
```

Ennek mintegy ellentettjeként lehet függvény az átadott paraméter, vagy annak összetevője, előállítója:

```
x = '13'
lebegopontos_szam = float
➥( int(x) )
```

## Saját függvénykísértéseink

A programozási nyelvek többsége lehetőséget ad arra, hogy a beépített függvényeken túl magunk is írhassunk ilyen konstrukciókat. ***Python***ban általában a következő formájúak a modulok tagfüggvényei:

```
modulneve.függvénye
➥(argumentum1, argumentum2, …)
```

A magunk készítette eszközöket viszont a ***def*** kulcsszóval kell bevezetni, és általánus alakjuk a következő:

```
def függvényneve (argumentum1,
➥argumentum2, … ):
  műveletek
  visszatérési érték
```

Ezek közül tetszés szerint elhagyható az argumentumlista (átadott paraméterek) és a visszatérési érték. Egyes nyelvek különbséget tesznek a függvények és az eljárások között; a ***Python*** esetén gyakorlatilag az eljárás olyan függvény, melynek nincs – pontosabban ***None*** értéket ad – visszatérési értéke.
Lényeges eleme a definíciónak a behúzás és az argumentumlista után található kettőspont; mindkettő azt jelzi, hogy a blokk feje a függvénymegnevezés, és az utána következő elemekkel együtt addig tart a függvény, amíg újra vele egy szinten álló elem nem következik (az értelmező hibát is jelez, ha elfeledkezünk a megfelelő tabulálásról).
A paraméterlistában szereplő argumentumok lehetnek állandók és változók egyaránt, leegyszerűsítve érték szerinti paraméter átadás történik (illetve referencia-szerű, adattípustól függően).
Ha valakinek szokatlan még a függvények használata, ügyeljen rá, hogy a definícióban szereplő paraméterek nevükben megegyezhetnek ugyan azokkal, melyeket utána a függvény törzsében láthatunk, azonban ez nem befolyásolja az eredeti, törzsblokkban szereplő változókat, mivel ott belül nem globális hatáskörrel rendelkező változókról van szó. Például:

```
#Függvény
def odszead(a,b):
  a = 3#lokális
➥hatókörű változó
  print 'A függvényen
➥belül a értéke:' , a
  c =  a+b
  return c
```

```
#Programtörzs
a=10
osszead(a,7)
print 'A programtörzsben
➥függvényhívás után is maradt
➥a értéke', a
```

Megadhatunk előre is alapértelmezett értékeket a függvények deklarálásakor, így ha nincs az adott helyen paraméter-átadás, akkor az alapértelmezett értékkel hívjuk meg a függvényt:

```
def fuggveny(megnevezes,
➥nap=13):
  print 'Ma ' ,
➥megnevezes, ',' , nap , '.'
➥,'van.'

fuggveny(megnevezes='péntek')
```

Fontos azonban a paraméterek sorrendje, ha az előző példában ezt írtuk volna:

```
def fuggveny(nap=13,
➥megnevezes):
```

akkor az értelmező hibát jelzett volna, hiszen az alapértelmezett értékkel bíró paraméter megelőzi az átadásra várót.

## Grafika a konzolon

Egyszerűbb grafikus megjelenítést igénylő feladatokban, ahol csak illusztrációként használjuk a képernyő adta lehetőségeket, kézenfekvő megoldást jelent a konzolablakon belül megjeleníteni programunk kimenetét. Erre kínál megoldást ***Unix*** rendszerek alatt általában a ***curses***, míg (***GNU) Linux*** disztribúciók alatt az ***ncurses*** függvénykönyvtár. Használata tehermentesít minket a különböző terminálok közötti különbségek okozta bosszúságoktól, és standard elérést biztosít a képernyőhöz mint kimeneteli formátumhoz. Egy hasznos segédlet a ➲ ***http://www.amk.ca/python/howto/curses/*** címen található.
A függvénykönyvtár importálása után annak `initscr()` függvény megállapítja az aktuálisan használt terminál típusát, és sikeres inicializálás esetén a teljes képernyőt képzeletben leképező objektumot adja vissza:

```
import curses
kepernyo = curses.initscr()
```

Előfordulhat, hogy nem elégszünk meg a képernyőt képzeletben leképező egyetlen ablakobjektum adta lehetőségekkel, és szeretnénk több, kisebb virtuális ablakot is használni. Mindezt megtehetjük a ***newwin(magasság, szélesség, balfelső_sarok_y_koordináta, balfelső_sarok_x_koordináta)*** fügvénnyel, ahol a képzeletbeli koordinátarendszer origója (y=0 ; x=0) a képernyő bal felső sarkában található. Ha például két, 10x10-es négyzet alakú képzeletbeli ablakot szeretnénk egy 80x25 karakter felbontású terminálablakban, s az egyiket a képernyő bal felső, a másikat a jobb felső sarkához illesztve kérjük, a következőt kell tennünk:

```
ablak_balfelaso =
➥curses.newwin(10,10,0,0)
ablak_jobbfelso =
➥curses.newwin(10,10,0,70)
```

Ha csak ennyit írunk programunkba, a lefuttatás után nagy eséllyel láthatatlanná válik a kurzor, és a parancsértelmező ugyan végrehajtja az utasításokat, de kifejezetten kellemetlen élmény vakon gépelni. Éppen ezért a feladat végrehajtása után vissza kell adnunk a képernyővezérlést a parancsértelmezőnek, azaz meg kell hívni a ***curses*** könyvtár használatát jelző függvényt:

```
curses.endwin()
```

Egyszerűbb karakteres grafikákat tudunk alkotni a `curses.vline()` és `hline()` függvényei segítségével, melyek paraméterezésére mutat példát a következő rövid program:

```
#!/usr/bin/python
import curses
kepernyo = curses.initscr()
kepernyo.erase()       #töröljük
➥a képernyő tartalmát
aktivterulet = curses.newwin
➥(1,1,25,80)    #képzeletbeli
➥aktív ablak

kepernyo.vline(5,10,'*'10)
kepernyo.hline(1,10,'*',5)

kepernyo.refresh()     #ez
➥szükséges ahhoz, hogy lássuk
➥az eredményt

curses.endwin()
```

A `vline()` paraméterezése úgy történik, hogy a rajzolandó függőleges „vonal” bal felső sarkának y, majd az x koordinátáit adjuk meg, majd azt a karaktert – esetünkben a csillagot –, mellyel „vonalat” szeretnénk húzni, végül a vonal hosszát. A `hline()` hasonlóképp, viszont itt vízszintes vonal a végeredmény. A `curses.refresh()` metódusra azért van szükség, mert hagyományosan a terminálokon nem csak a mai értelemben vett képernyős megoldásokat, hanem soros, modemes vagy egyéb, lassú vonalon érkező adatforgalmat értettek, s nem volt mindegy, milyen sűrűn kell frissíteni a tartalmat. Ahhoz, hogy lássuk is ügyködésünk eredményét, általában a `refresh()`-t, vagy ehhez hasonló szerepű függvényt kell igénybe vennünk.

## Rekurzív tornyosulás

A programozás során adódhatnak olyan helyzetek, amikor a fejlesztőnek el kell döntenie, hogy átlátható de esetleg erőforrás-igényesebb kódot, vagy optimalizált, de kevésbé magától értetődő programot kell készítenie. Inkább előbbire jó példa a rekurzió, mert elsődlegesen az ember, s nem a gép munkáját könnyíti meg.
A rekurzió olyan vezérlési módnak tekinthető, mely során a függvény saját magát hívja meg újra és újra, egészen addig, míg a leállításáért felelős feltétel nem teljesül. Egyszerű példája: egy teremben sokan ülnek, és szeretnék megtudni, pontosan hányan vannak. A számolás algoritmusa egyesével haladva, ha mindenki a szomszédjának adja tovább az eredményt a következő:

***az első ember mondja a szomszédjának azt, hogy 1;***
***ha te vagy az utolsó ember,*** {
  ***akkor állj meg, adj hozzá egyet a szomszédod által kapott számhoz, és mondd el, mit kaptál,***
}
***egyébként*** {
***fordulj a szomszédodhoz, és adj hozzá egyet az eddig kapott számhoz***
}
***addig ismételjétek, amíg igaz a feltétel.***

Ugyanennek klasszikus példája a 19. századból származó, ***Hanoi tornyai*** legendává vált feladvány, mely szerint

a szerzetesek egy feladatot kaptak, s annak végrehajtása után jön el a világvége. A feladat egyszerű: adott három rúd, az elsőn 64 darab alulról felfelé egyre csökkenő átmérőjű középen lyukas korong. Ezt kell úgy átpakolni a harmadik rúdra, hogy használhatják segédletként a második rudat, de a korongok mindig csak úgy állhatnak, hogy nagyobb átmérőjű ne álljon kisebb átmérőjű tetején, és egyszerre csak egy korongot lehet mozgatni. Egyszerűsítsük tovább dolgunkat azzal, hogy egyelőre csak három koronggal foglalkozunk, ugyanis $2^{64}$-1 lépés szimulálása még számítógéppel is kissé sokáig tartana.
Az elegáns rekurzív megoldáshoz úgy juthatunk el, ha a problémát három fázisra bontjuk:

1. a legnagyobb korong kivételével minden korongot rakjunk át a középső oszlopra ideiglenes (ami azt is jelenti, hogy itt már rendezetten fognak állni, felülről lefelé növekvő átmérőjű korongok lesznek)

2. a legnagyobb korongot rakjuk át végleges helyére, az üres harmadik oszlopra

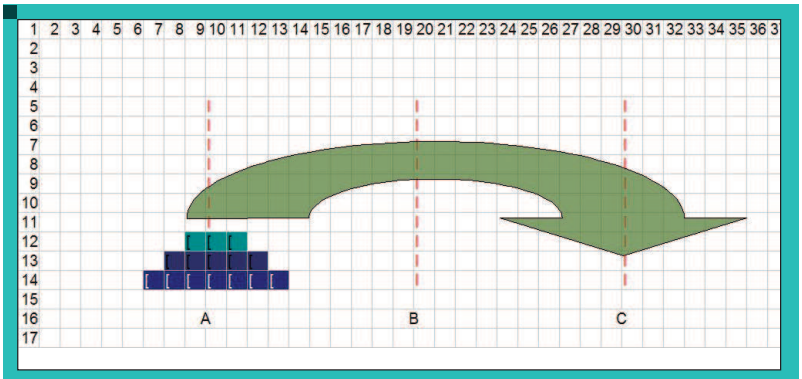
3. a középső oszlopon álló korongokat is rakjuk át megfelelő sorrendben a harmadik oszlopra

Megfigyelhetjük, hogy a harmadik fázisból ugyanazt az ismétlődő, önmagát újra és újra meghívó tevékenységsorozatot tudjuk felírni, egészen addig, míg csak egy korong marad, annak átpakolása pedig már egyértelmű. Mindez Pythonban körülbelül így néz ki:

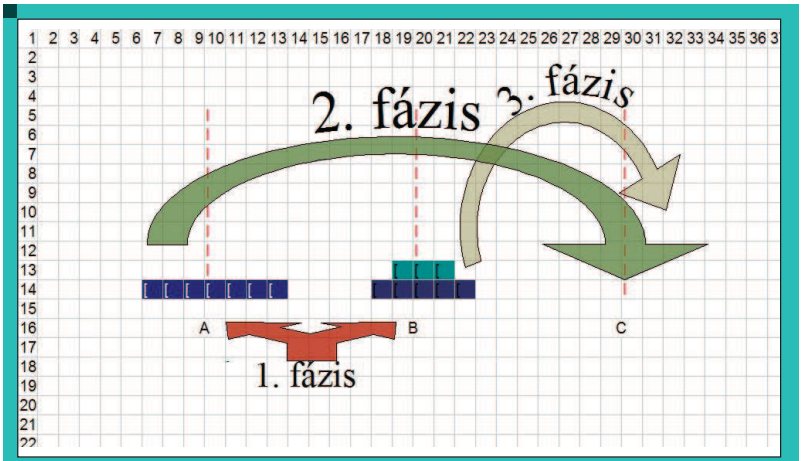
```
def Hanoi(darab, honnan, hova,
➥ideiglenes):
  if darab > 0:
    Hanoi(darab-1, honnan,
➥ideiglenes, hova)
    print honnan , '-->', hova
    Hanoi(darab-1, ideiglenes,
➥hova, honnan)

Hanoi(3, 'A', 'C', 'B')  #A,B,C
➥oszlopok sorban haladva
```

Ezt lefuttatva láthatjuk a lépéseket. Mindeközben a számítógép folyamatosan meghívja ugyanazt




**1. ábra** Hanoi elképzelt tornyai



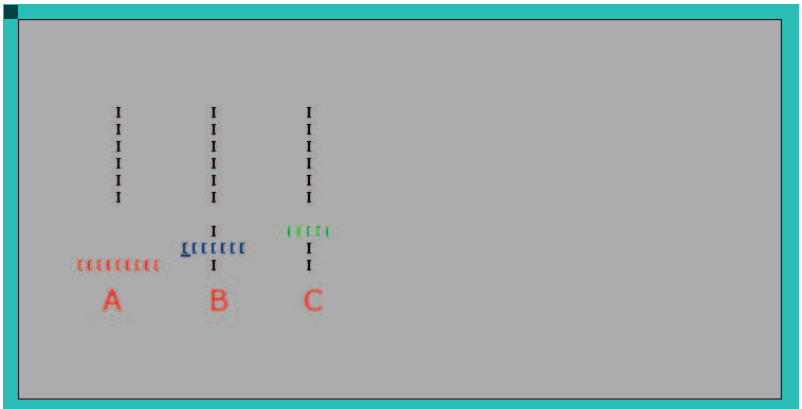

**2. ábra** A három fő lépés

a függvényt, de a feltétel miatt (`if darab>0`) nem esik végtelen ciklusba, hiszen a folyamatosan csökkenő argumentumérték (darab változó) nem hagyja. A függvények ugyanúgy leegyszerűsítik a problémát, mint mi tettük a három fázissal, hiszen A , B, C oszlopként hivatkoztunk a három rúdra, és egészen addig csökkentettük a darabszámot (azaz vettük le képzeletben sorban a korongokat), amíg az azonnal megoldható, már csak egy korongot tartalmazó rúdról kellett végleges helyére rakni az adott korongot.
Használjuk fel a *curses*-szel való ismerkedés során szerzett csekély tapasztalatunkat, és próbáljuk meg kicsit jobban láthatóvá tenni ugyanezt a megoldást. Ehhez csak csekély mértékben kell a forrást módosítani, leginkább kibővíteni azt. Nem törekszünk egyelőre elegáns megoldásra, inkább a viszonylag követhető utat próbáljuk meg bejárni, s majd ha jobban megbarátkoztunk az objektumokkal, térjünk vissza ismét egy szebb programmal.

```
#!/usr/bin/python
#Curses alapú struktúrált,
➥rekurzív program Hanoi
➥tornyai megoldására
import curses
kepernyo =  curses.
➥initscr()    #konzolos
➥grafika inicializálása

aktivterulet = curses.newwin
➥(1,1,25,80)  #kb. egy
➥képernyőnyi terület

#Rúd és korongok
#Első rúd (10;5) - (10;15)
kepernyo.vline(5, 10, 'I', 10)
#Második rúd (20;5) - (20;15)
kepernyo.vline(5, 20, 'I', 10)
#Harmadik rúd (30;5) - (30;15)
kepernyo.vline(5, 30, 'I', 10)
```




**3. ábra** Curses+képzelet=Hanoi

```
#Korongok kezdetben első rúdon
➥- 4 db
kepernyo.hline(11, 9 , '[', 3)
kepernyo.hline(12, 8 , '[', 5)
kepernyo.hline(13, 7 , '[', 7)
kepernyo.hline(14, 6 , '[', 9)

curses.napms(1200)   #szándékos
➥késleltetés
kepernyo.refresh()

#Hanoi függvény - rekurzió
def Hanoi(meret, honnan, hova,
➥ideiglenes):
  if meret > 0:
    Hanoi(meret-1, honnan,
➥ideiglenes, hova)
    torles(honnan, meret)
    atrak(hova, meret)
    Hanoi(meret-1, ideiglenes,
➥hova, honnan)
    torles(honnan, meret)
    atrak(hova, meret)

#Törlés függvény
def torles(honnan, meret):
  if honnan == 'A':
    kepernyo.hline(10+meret,
➥10-meret, ' ' , meret*2+1)
➥#felülírjuk háttérszínű
➥szóközzel
    curses.napms(600)
    kepernyo.refresh()
  elif honnan == 'B':
    kepernyo.hline(10+meret,
➥20-meret, ' ' , meret*2+1)
➥#felülírjuk háttérszínű
➥szóközzel
    curses.napms(600)
    kepernyo.refresh()
  else:
    kepernyo.hline(10+meret,
➥20-meret, ' ' , meret*2+1)
➥#felülírjuk háttérszínű
➥szóközzel
    curses.napms(600)
    kepernyo.refresh()

#Rajzolás függvény
def atrak(hova, meret):
  if honnan == 'A':
    kepernyo.hline(10+meret,
➥10-meret, '[' , meret*2+1)  #[
➥jelekből rajzolunk
    curses.napms(600)
    kepernyo.refresh()
  elif honnan == 'B':
    kepernyo.hline(10+meret, 20-
➥meret, '[' , meret*2+1)     #[
➥jelekből rajzolunk
    curses.napms(600)
    kepernyo.refresh()
  else:
    kepernyo.hline(10+meret,
➥20-meret, '[' , meret*2+1)  #[
➥jelekből rajzolunk
    curses.napms(600)
    kepernyo.refresh()

#Programtörzs
Hanoi(4, 'A', 'C', 'B')
#függvényhívás, rekurzió
➥elindítása

curses.endwin()    #konzolos
➥grafika vége
```

Első ránézésre feltűnhet, hogy ez bizonyosan nem optimális megoldás, hiszen sokszor szinte ugyanazokkal a paraméterekkel állítunk elő újabb rajzokat. A rajzolás és törlés különválasztásának elsősorban azért volt értelme, mert így jobban kivehető, hogyan is zajlik a mozgatás illúziója: először beazonosítjuk, melyik korongról indul a mozgatás (honnan paraméter), majd a megfelelő korongot háttérszínű szóközökkel feltöltve eltüntetjük. Jobban mutatna, ha ügyelnénk arra is, hogy a szóközök közepére visszarajzoljuk a rudat szimbolizáló [ jelet is.

A korong beazonosítása már valamivel intelligensebb megoldás, kihasználjuk azt a tényt, hogy a rekurzív függvényhívások során a korongok darabszáma (azaz a magassága) egyben a korong méretéről is ad információt: 1 a legkisebb – és alapesetben legfelső, 2 a középső, és 3 a legnagyobb korongot jelenti. A többi már csak egy négyzetrácsos papír és némi számolgatás kérdése. A gyors működés miatt került bele a késleltetés (`curses.napms()`, milliszekundumokban mérve a szünetet), így láthatóvá vált, hogyan mozognak a tárgyak.

Szebb megoldást jelent már az is, ha egy ciklussal képezzük a korongokat, például így:

```
for i in range(darab, 0, -1):
  kepernyo.hline(5+i, 10-1,
➥'[', i*2+1)
  kepernyo.refresh()
```

A törlés és rajzolás függvény is összeolvasztható egy komplex mozgatófüggvénytörzsbe, és a feltételvizsgálatokon is lehet finomítani. De a legnagyobb előrelépést az jelenti, ha zavar minket a gravitáció hiánya (a korongok jelenleg csak vízszintesen mozognak, függőlegesen nem, így nem reálisan helyezkednek el a rudakon ), és ezért átírjuk vagy a korongok magasságának figyelésére, vagy az adott rúd korongjainak sorba állításával – főleg háromnál több koronggal látványos – akár egy ismertebb rendezőalgoritmus segítségével.

*Tóth Virgil Zoltán*
(m_v@c2.hu)
Szoftverfejlesztő informatikus és rendszergazda, kedvence a Debian disztribúció. Szabadidejét legszívesebben felesége és szépirodalmi regények társaságában tölti. Lenyűgözőnek tartja a Linux rugalmasságát, és a vele dolgozók aktivitását.