© Kiskapu Kft. Minden jog fenntartva

Bevezetés a Tkinter használatába (4. rész)

Záróakkordok…

Bevezetésünk végéhez érkeztünk. Mostanra már tudjuk, hogyan néz ki egy Tkinter-program és megismerkedtünk a rendelkezésünkre álló lehetőségekkel. Ha figyelmesen követtük a cikksorozat eddigi részeit, és olykor a leírást is elővettük, valószínűleg már magunk is képesek vagyunk egyszerű programok írására.
Sorozatunk elkövetkező részeiben zavarosabb vizekre, ismeretlen tájak felé evezünk, vagyis komolyabb témákkal foglalkozunk. Szolgáljon ez a rész hídként, amellyel a már ismertet az ismeretlennel összekötjük!
Minden program egyik legfontosabb jellemzője a sebessége. Különösen igaz ez az olyan programokra, amelyek folyamatos felhasználói beavatkozást igényelnek, egy kérdőívhez olvasnak be adatokat, képeket jelenítenek meg, vagy éppen kiszámolnak valamit.
Az ilyen programoknál a felhasználó elvárja, hogy miután leüt egy billentyűt vagy megnyom egy gombot az alkalmazás ablakában, azonnal valamilyen válasz következik be: megjelenik az adott betű, beugrik valamilyen új ablak – tehát a program valamiféle jelét adja annak, hogy bizony működik. A felhasználó sokszor a program e viselkedése alapján dönt két alkalmazás között, és előfordulhat, hogy azt ítéli meg jobbnak, amelyik minden eseményre azonnal reagál, még ha a kiválasztott program az adott dolgot rosszabbul végzi is el.
Képzeljük el, hogy szöveget gépelünk be egy szövegbeviteli mezőbe, netán a beviteli mezők között ugrálunk a TAB gombbal. Kevés dolog zavaróbb annál, mintha a betűk nem jelennek meg azonnal, ahogy beírjuk őket. Olyan esettel is találkoztunk, hogy űrlapok sorozatát kellett kitöltenünk, és minden egyes űrlap után hosszú másodperceket kellett arra várnunk, hogy a következő ablak betöltődjön.
A most következőkben néhány olyan trükkel ismerkedünk meg, amelyekkel alkalmazásainkat gyorsabbá, hatékonyabbá és kezelhetőbbé tehetjük.

Kapcsoljunk sebességbe!

Ha már programindításnál a felhasználó kedvében szeretnénk járni, ne várakoztassuk meg feleslegesen! Az egyik legtöbb időt igénylő folyamat a Python-kód bájtkódra történő lefordítása, mely végül ténylegesen futni fog. Ha programunk sok-sok ezer sort tartalmaz, ez bizony nem kevés időt vesz igénybe. Viszont tudjuk, hogy a rendszer moduljainkat önműködően bájtkódra fordítja le és tárolja is őket ilyen formában, ***.pyo*** vagy ***.pyc*** kiterjesztéssel. A következő futáskor – ha a bájtkód újabb, mint a ***.py*** kiterjesztésű fájl – anélkül, hogy az eredeti kóddal bármit kezdene, a Python azonnal a bájtkódhoz fordul. Programunk indítófájlját a Python ugyanakkor lefordítja, de a bájtkódú állományt nem tárolja. Ez nem éppen szerencsés, tekintve ha alkalmazásunk logikája és a temérdek kód ebben az indítófájlban van tárolva. Nincs más teendőnk, mint ezt a fájlt modullá alakítani, és létrehozni egy néhány soros fájlt, ami ezt a modult importálja.
Miről is van szó? Tegyük fel, hogy van egy alkalmazásunk, mely a következő programsorokat tartalmazza:

```
from Tkinter import *
import Pmw

class EgyOsztaly(Dialog):
        sok ezer sor

root = Tk()
peldany = EgyOsztaly(root)
```

Ha ezt a fájl közvetlenül a Pythonnal hívjuk meg, a Python értelmezője minden egyes alkalommal bájtkóddá fordítja le, és csak azt követően hajtja végre.
Alakítsuk át a program utolsó két sorát függvénnyé, és szúrjuk be eléjük a `def indito:` sort! Ha most ezt az `indito` eljárást egy másik indítófájlból hívjuk meg, akkor ezt a sok ezer kódsort tartalmazó modult máris csak egyszer, a legelső alkalommal fordítja le nekünk a Python, és az eredményt bájtkódba menti!
Az új indítófájlunk így néz ki:

```
import RegiIndito
RegiIndito.indito()
```

És máris rengeteg időt takarítottunk meg, mert a Pythonnak minden alkalommal csak két sort kell lefordítania, a többi már a bájtkód formátumú fájlokból töltődik be.
Ha a Pythont már első indításkor az `-O` kapcsolóval hívjuk meg, programunk még gyorsabb lesz, mivel a Python egyszerűsíti nekünk a kódot, illetve a bájtkódba nem fordít bele olyan dolgokat, amelyekre futáskor nincs szükség (például nem tárolja minden művelethez kapcsolódóan, hogy eredetileg a forráskód melyik sora tartalmazza azt). Az ilyen módon fordított modulok ***.pyo*** kiterjesztést kapnak, mely a ***.pyc*** fájlokhoz hasonlóan a futtató rendszertől úgyszintén függetlenek.

Egyszerűsítsünk!

A legfontosabb szabály, hogy programunkat úgy tervezzük meg, hogy a Python értelmezője minél kevesebb időt töltsön programunkban, és inkább a saját belső függvényeiben tengesse az idejét. Ez annyit tesz, hogyha egy műveletre létezik valamilyen Python-függvény, azt részesítsük előnyben. Ha mindent saját magunk kódolunk, a ciklusok futása során értékes időt veszítünk, ezért jobban tesszük, ha a Python C-ben írt függvényeire bízzuk magunk.
Gyakori eset, hogyha egy adott függvényben egy külső változóra van szükség, a függvény elején a külső változót egy helyi változónak adjuk értékül. Ez viszonylag egyszerű művelet, de ha a függvényünk mondjuk, valamilyen ciklusból hívódik meg, amely akár több ezerszer végrehajtódik, akkor ilyen apróságoknak is érdemes figyelmet szentelnünk. Így mielőtt helyi változót hoznánk létre, gondoljuk meg, valóban szükség van-e rá. Ha a helyi változóhoz a függvényből csak egyetlen alkalommal férünk hozzá, nem érdemes rá értékes órajelciklusokat pazarolni. Ellenben ha a függvényünkben többször hivatkozunk rá, netalántán a függvény egy újabb ciklust tartalmaz, akkor jobban tesszük, ha létrehozzuk azt a helyi változót. Ilyen módon elérhetjük, hogy a külső

változót csak egyszer kelljen elérni, azután pedig használhatjuk a jóval gyorsabban elérhető helyi változónkat.
Vannak esetek, amikor megszokásból létrehozunk változókat, amiket aztán soha többet nem használunk, vagy csak egyetlenegyszer. Ilyen helyzetben mindig tegyük fel magunknak a kérdést: hogyan oldható meg a dolog egyszerűbben? Ha például egy `Label()` objektumot hozunk létre, amelyre csak egyszer hivatkozunk – amikor kitesszük a képernyőre –, akkor jobban tesszük, ha változó megalkotása helyett a Pythonra bízzuk a dolgot, és egy választékos `Label().pack()`-kel oldunk meg mindent.
Ha mégis ciklusok írására adjuk a fejünket, gondoljuk át még egyszer, nincs-e az már adott feladatra belső függvény… Biztos nem megoldható a dolog se a `map()`-pel, se a `reduce()`-szal, se a `filter()`-rel? Ha nem, hát nem.
Vegyünk egy példaciklust:

```
for i in range(10000):
    for j in range(100):
      import EgyModul
      a = a + EgyModul.EgyOsztaly.EgySzam * 2
```

Ez a ciklus tízezer alkalommal egy százelemű listát hoz létre, 10 000-szer 100 alkalommal importál egy modult, és ugyanennyiszer fér hozzá egy külső változóhoz. Ha a fenti kódot picit átírjuk, jelentős sebességnövekedést érhetünk el:

```
import EgyModul
b = EgyModul.EgyOsztaly.EgySzam

szaz = range(100)

for i in range(10000):
    for j in szaz:
        a = a + b * 2
```

A második kód nyolcszor gyorsabb az elsőnél! Lássuk, hogyan értük el: a második esetben a modul csupán egyetlenegyszer importálódik, a százelemű listát egyszer hozzuk létre, és külső változót is egyszer olvasunk be. Ha az `a` kezdőértékének nullát veszünk, az `EgyModul.EgyOsztaly.EgySzam` értékének pedig hármat, akkor mindkét esetben hatmilliót kapunk végeredményként. Láthatjuk tehát, hogy egyáltalán nem mindegy, hogyan oldunk meg egy nehézséget – akár egy egyszerű számolást is.
Vegyünk két kifejezést:

```
szoveg = szoveg1 +      + szoveg2 +      + szoveg3
```

és

```
szoveg =  %s %s %s  % (szoveg1, szoveg2, szoveg3)
```

A második esetben – C-kódban gondolkodva – az egész egy egyszerű függvénnyel megoldható, míg az első esetben memóriaműveletek sorára van szükség. Így már nem olyan meglepő, hogy az utóbbi megoldás közel ötször gyorsabb az elsőnél.
Egyetlen szabály van tehát, amit fejben kell tartanunk, mégpedig az, hogy a kódunk minél rövidebb és egyszerűbb legyen. Gondolkodjuk a Python „fejével”, és amit csak lehet, bízzunk rá.

Tegyünk úgy, mintha gyorsak lennénk!

Mint már szó volt róla, grafikus felület tervezésénél a felhasználó számára a legfontosabb annak látszata, hogy programunkat gyorsnak lássa. Ha a képernyő előtt ülve napjában többször is űrlapok során kell végigverekednie magát, nem mindegy, mennyit kell várnia az egyes űrlapok között. Az időigényes műveleteket lehetőség szerint úgy kell csoportosítani, hogy a felhasználónak inkább egyszer kelljen hosszabb ideig várnia.
Ha a felhasználót sokszor váratjuk, programunkat lassúnak fogja érzékelni, míg ha csak a kitöltés befejeztével kell várnia, ő már léphet is tovább a következő munkájára, programunk pedig a háttérben dolgozik.
Figyeljünk arra is, hogy az olyan műveleteket végképp kerüljük el, melyeket a grafikus felület kirajzolásakor hajtanánk végre. A lassan, akadozva felbukkanó ablakok a felhasználóban azt az érzést keltik, hogy a program hibásan működik.
Hasznos, ha az elemeinket tartalmazó kereteket (Frame) csak azután tesszük ki a képernyőre, miután a keretben található elemek elkészültek. Ha fordítva járunk el, a felhasználó esetleg villogást tapasztal – ez annak következménye, hogy az elemkezelő minden új elem kihelyezésekor a többi helyzete alapján számolásokat végez, hogy az új hova is kerüljön.
A felhasználó tényekedése által keltett eseményekre a lehető leggyorsabban próbáljunk meg válaszolni. A puszta egérmozgatás is események egész sorát indítja el, ami – ha csak az egérmutató koordinátáira van szükségünk – nem is baj, de ha egy ilyen gyakran keletkező eseményhez olyan függvényt rendelünk, amely bonyolult elemet rajzol ki, akkor ennek a rendszer teljesítményére nézve súlyos következményei lesznek.
Azért erre is van megoldás: ha mi mégis ilyen bonyolult műveletet szeretnénk elvégezni, akkor lehet, hogy elegendő minden 5. eseményre válaszolunk, vagy meghatároznunk, hogy a kirajzolás legfeljebb csak bizonyos időközönként hajtódjon végre, a többi esetben pedig egyszerűen hagyjuk figyelmen kívül az eseményt.

Keressük meg a szűk keresztmetszetet!

Ha programunk futása során úgy érezzük, hogy mozgása minden igyekezetünk ellenére mégis döcögős, és nem tudjuk, melyik függvényben időzik el, jó szolgálatot tehet a `profile` modul. E remek kis modul segítségével könnyen megtalálhatjuk, hol akad el programunk futása.
Amennyiben programunk egy `indito()` függvény meghívásával kezdi el a munkát, annyit kell tennünk, hogy programunk elejét a következőképpen alakítjuk át:

```
import profile
profile.run( indito() )
```

Miután ez lefutott, a profiler megjeleníti a statisztikát, hogy programunk hány másodpercet töltött el az egyes függvényekben.
Ez a statisztika szedett-vedett, tegyük kicsit rendbe:

```
import profile
import pstats

profile.run( indito() ,  eredmenyek.p )
p = pstats.Stats( eredmenyek.p )
p.sort_stats( cumulative ).print_stats(10)
```

Ebben az esetben a **profile.run()* létrehoz egy ***eredmenyek.p*** nevű fájlt, mely a profiler kimenetét tartalmazza a `pstat` modul által emészthető formában. Ezt követően a fájlt a `Stats()` függvénnyel beolvassuk, és megjelenítjük a tíz legtöbb időt igénybevevő függvényt.

Gludovátz Gábor
(ggabor@sopron.hu) Kedvenc időtöltéseinek egyike a programozás és a Linux lelkivilágának alaposabb megismerése. Gyakran éjszakázik a monitor előtt ülve hosszú kódsorok társaságában, és ha ideje engedi, a soproni erdőkben teker kedvenc bringájával.

© Kiskapu Kft. Minden jog fenntartva