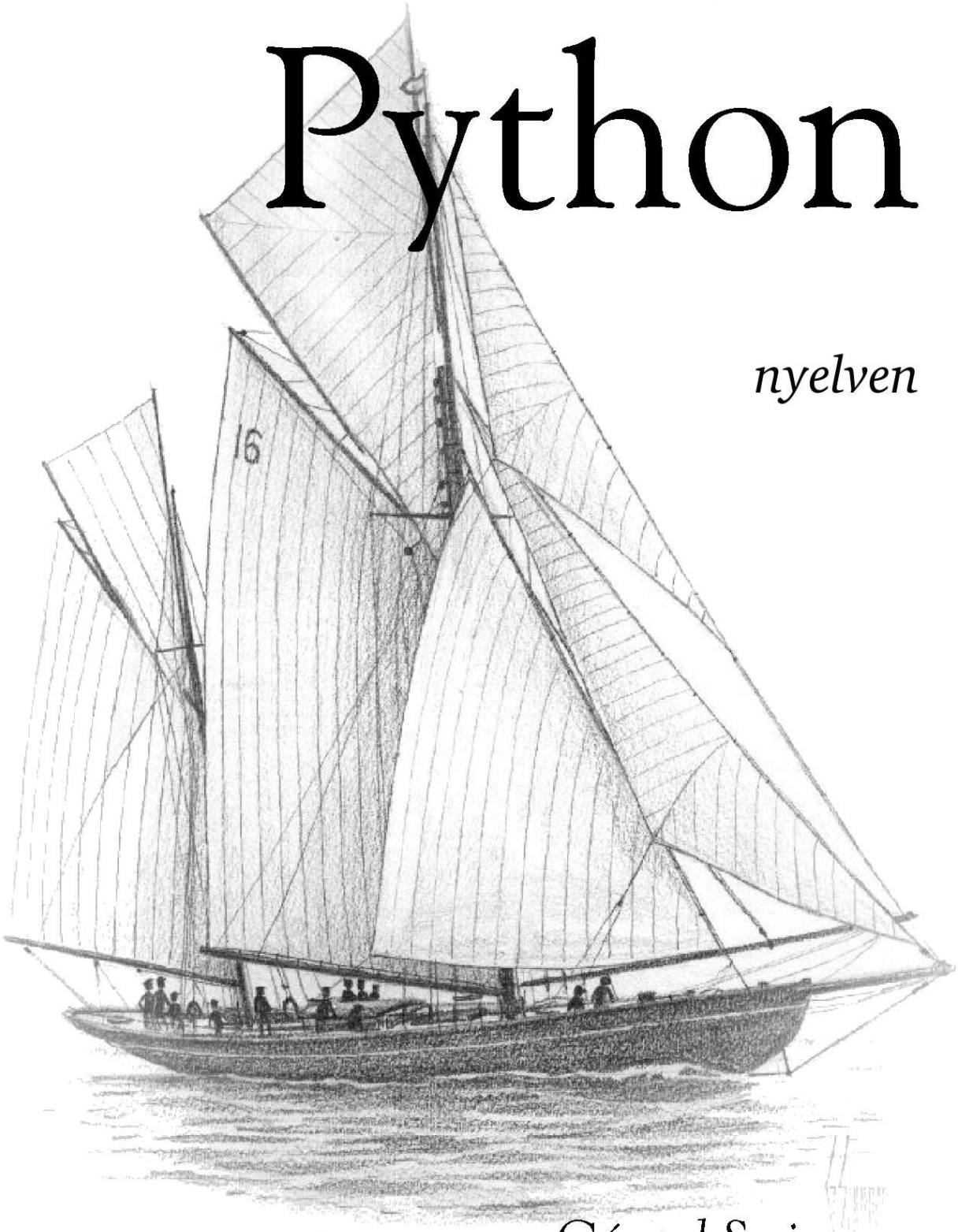
Tanuljunk meg programozni

Python

nyelven



Gérard Swinnen

Allen B. Downey, Jeffrey Elkner & Chris Meyers
"How to think like a computer scientist" művének szabad adaptációja

Grace Hopper, a compiler feltalálója :

« Számomra a programozás több mint alkalmazott tudomány. A programozás a tudás mélységeibe vezető érdekes kutatás is. »

Maximiliennek, Élisenek, Lucillenek, Augustinnek és Alexanenak.

Colophon

A borító grafitrajzát, aminek tárgya szádékosan tér el a könyv tárgyától, a szerző készítette 1987-ben egy régi fénykép alapján Canson papírra. A cowes-i kikötőben 1923-ban egy regattán résztvevő 106 tonnás *Valdora* versenyjachtot ábrázolja.

A húsz évvel korábban épített, és eredetileg yawl-nak felszerelt *Valdora* több díjat elhozott, mielőtt 1912-ben a rajzon látható 516 m² vitorlafelületű ketch-é átalakították.

Ez a pompás vitorlás, amit legénysége igen nagyra becsült kiváló tengerjáró tulajdonságaiért, majdnem fél évszázadon át közlekedett.

Tanuljunk meg programozni Python nyelven

Gérard Swinnen
tanár és pedagógiai tanácsadó
Institut S[t] Jean Berchmans - S[te] Marie
59, rue des Wallons - B4000 Liège

Ez a jegyzet szabadon letölthető a következő website-ról :
http://www.ulg.ac.be/cifen/inforef/swi

A szöveg egy részét a következő mű inspirálta :
de Allen B. Downey, Jeffrey Elkner & Chris Meyers
How to think like a computer scientist

megtalálható : http://rocky.wellesley.edu/downrey/ost
vagy : http://www.ibiblio.org/obp

Bevezetés

Ez a jegyzet eredetileg a belga középfokú oktatásban résztvevő, *Tudomány és informatika* fakultáción *Programozás és programnyelvek* tantárgyat tanuló 3. osztályosok számára készült. Egy kísérleti szöveg, amit az *interneten* szabad licenc alatt publikált számos más dokumentum nagymértékben inspirált.

A jegyzetben egy nem lineáris tanítási eljárást javaslunk, ami biztos hogy kritizálható. Tudatában vagyunk annak, hogy ez egy kicsit kaotikusnak fog tűnni egyes puristák szemében, de mi akartuk így, mert meg vagyunk róla győződve, hogy többféle (nemcsak programozás, hanem egyéb) tanítási módszer létezik és el kell fogadni azt a tényt, hogy a különböző emberek ugyanazokat a fogalmakat nem ugyanabban a sorrendben tanulják meg. Mindenekelőtt a figyelemfelkeltésre törekedtünk és arra, hogy a következő vezérelvek betartásával a kapukat szélesre tárjuk :

- A tanításnak egy átlagos diák értelmi szintjéhez és általános ismereteihez kell alkalmazkodni. Azt elutasítjuk, hogy kis zsenikből álló « elit »-nek oktassunk. Ezen a szemüvegen át nézve a dolgokat az az általános törekvésünk, hogy bármiféle specializáció nélkül nyilvánvalóvá tegyük a programozás és az informatika invariánsait.
- A tanítás során alkalmazott eszközöknek korszerűeknek és versenyképeseknek kell lenni, de az is szükséges, hogy a diákok személyes használatra legálisan jussanak hozzájuk. Tanítási módszerünk azon az elképzelésen alapul, hogy a tanulóknak nagyon korán el kell kezdeniük saját projektjeik megvalósítát, amiket a saját elképzeléseik szerint fejlesztenek és használnak fel.
- A tanulónak nagyon hamar képesnek kell lenni kis grafikus alkalmazások megvalósítására. Nagyon fiatalokhoz szólunk (elvileg éppen abba a korba érkeztek, amikor képesek elkezdeni absztrahálni). Amellett foglaltunk állást, hogy nagyon korán térjünk rá a grafikus interface programozásra, még a rendelkezésre álló adatszerkezetek bemutatása előtt, mert megfigyelhető, az osztályainkba érkező fiatalok már egy ablakokon és más interaktív grafikus interface-eken alapuló informatikai kultúrában « lubickolnak ». Ha a programozás tanulást választják, természetesen szeretnének minél előbb (lehet, hogy nagyon egyszerű) alkalmazásokat készíteni, melyekben a grafikus megjelenés már masszívan jelen van. Azért választottuk ezt a kissé szokatlan megközelítést, hogy növendékeinknek nagyon korán lehetőségük legyen kis, személyes projektekbe kezdeni, melyek révén érezhetik, hogy értékelik őket. Azt viszont megköveteljük, hogy a munkáikat automatikus kódgeneráló, fejlett programfejlesztő környezetek alkalmazása nélkül írják meg, mert nem akarjuk a programozás összetettségét sem elfedni.

Egyesek azzal kritizálnak bennünket, hogy módszerünk nem állítja eléggé középpontba a tiszta és kemény algoritmizálást. Úgy gondoljuk, hogy egy ilyen megközelítés - a fentebb már említett okok miatt - nincs a fiatalokra adaptálva. Ráadásul ez a megközelítés kevésbé lényeges, mint a múltban volt. Úgy tűnik, az objektumokkal történő modern programozás tanulása inkább azt igényli, hogy a tanuló amilyen korán csak lehet kerüljön kapcsolatba már létező objektumokkal és osztálykönyvtárakkal. Így nagyon korán megtanulja, hogy inkább objektumok közötti interakciókban gondolkozzon, mint eljárásokban és ez lehetővé teszi, hogy elég gyorsan kiaknázza az olyan konstrukciók előnyeit, mint az öröklés és a polimorfizmus.

Egyébként elég jelentős teret biztosítottunk a különböző típusú adatszerkezetek kezelésének, mert úgy véljük, hogy az adatszerkezetek átgondolásának kell képezni minden programfejlesztés gericét.

Az első programozási nyelv kiválasztása

Sok programozási nyelv létezik. Mindegyiknek vannak előnyei és hátrányai. Biztos, hogy az lenne az ideális, ha több nyelvet használnánk. Csak bátorítani tudjuk a tanárokat, hogy szánjanak rá időt és mutassanak be különböző nyelvekből vett példákat. El kell azonban fogadnunk, hogy mindenek előtt a szilárd alapok megszerzésére kell törekednünk és a rendelkezésünkre álló idő korlátozott. Ezen a szemüvegen át nézve a dolgokat az tűnik észszerűnek, hogy ha először csak egy nyelvet használunk, legalább is az első tanévben.

De melyik nyelvet válasszuk elsőnek ?

Amikor az új Tudományok és Informatika tantárgy tantervének előkészítése során elkezdtünk gondolkodni ezen a kérdésen, már elég hosszú személyes programozási tapasztalatot gyűjtöttünk össze *Visual Basic-kel (Micro$oft)* és *Clarion-nal* (*Top$peed*). *Delphi-vel* (*Borl@nd*) is kísérleteztünk egy kicsit.
Természetes volt, hogy először ezeknek a nyelveknek az egyikére gondoltunk (a *Clarion*-t preferálva, ami sajnos kevéssé ismert).

Ha ezeket akarnánk alapeszközként használni az általános programozás tanításához, akkor azok két komoly hátránnyal járnának :

- Az üzleti szoftverekhez tartoznak.
  Ez azt jelenti, hogy nemcsak az ezeket a szoftvereket használni kívánó oktatási intézménynek kellene minden munkaállomás számára licenet vásárolni (ami elég költségesnek ígérkezik), hanem azok a tanulók is implicit módon erre lennének kényszerítve, akik a programozási tudásukat az iskolán kívül kívánják alkalmazni, amit nem tudunk elfogadni.
- Speciálisan egyetlen operációs rendszerhez kötődnek, a *Windows* -hoz. Nem « portábilisak » más operációs rendszerekre (*Unix*, *MacOS*, stb.). Ez nem illeszkedik abba a pedagógiai tervünkbe, hogy általános (és ezért szerteágazó) képzést adunk, amiben az informatikai invariánsokat - amennyire lehetséges - nyilvánvalóvá tesszük.

Ezért úgy döntöttünk, hogy megnézzük az alternatív kínálatot, vagyis amit a szabad szoftver mozgalom[1] ingyen kínál. Amit találtunk, az nagyon fellelkesített bennünket : az *Open Source* világban ingyenes interpreterek és compilerek léteznek egy sor nyelvre, ráadásul ezek a nyelvek modernek, versenyképesek, portábilisak (azaz különböző operációs rendszerek alatt használhatók, mint a *Windows*, *Linux*, *MacOS* ...) és nagyon jól dokumentáltak.

Vitán felül a *C/C++* a domináns nyelv. Ezt a nyelvet abszolút referenciaként fogadják el és minden komoly informatikusnak előbb vagy utóbb össze kell vele akadni. A baj csak az, hogy nagyon ijesztő és bonyolult, túlságosan gépközeli. A szintaxisa kevéssé olvasható és erősen korlátozó. Egy nagyméretű program megírása C/C++ -ban hosszú és fáradságos. (Ugyanezek a megjegyzések nagymértékben érvényesek a *Java* nyelvre is).

1 A szabad software (*Free Software*) egy olyan program, aminek a forráskódja mindenki számára hozzáférhető (*Open source*). Gyakran ingyenes (vagy majdnem az), a szerzője szándéka szerint szabadon másolható és módosítható, általában a világ különböző részein élő lelkes, önkéntes fejlesztők százai együttműködésének a terméke. Mivel forráskódját számos specialista (egyetemi hallgató és oktató) « boncolgatta », ezért az esetek többségében a nagyon magas technikai színvonal a jellemzője. A leghíresebb szabad software a ***GNU/Linux*** operációs rendszer, aminek népszerűsége napról napra nő.

Másrészt ennek a nyelvnek a korszerű gyakorlata gyakran folyamodik alkalmazás generátorokhoz és más nagyon fejlett segédeszközökhöz, mint amilyen a *C++Builder*, *Kdevelop*, stb. Ezek a programfejlesztő környezetek nagyon hatékonynak bizonyulhatnak gyakorlott programozók kezében, de túlságosan sok összetett eszközt kínálnak és olyan ismereteket tételeznek fel a felhasználó részéről, melyekkel egy kezdő nyilvánvalóan még nem rendelkezik. Közöttük elvész a kezdő, nem látja a fától az erdőt, ami azzal a veszéllyel jár, hogy a segédeszközök elfedik a nyelv alapmechanizmusait. Ezért a C/C++ -t későbbre hagyjuk.

Programozási tanulmányaink elején számunkra preferálhatóbbnak tűnik egy magasabb szintű, kevésbé korlátozó, jobban olvasható szintaktikájú nyelv használata. Nézze meg az olvasó e tárgykörben Jeffrey Elkner « *How to think like a computer scientist* » könyvének előszavát (lásd a 362. oldalt).

Miután megvizsgáltuk és egy kicsit kísérleteztünk a ***Perl*** és ***Tcl/Tk*** nyelvekkel, végül a nagyon korszerű és növekvő népszerűségnek örvendő ***Python*** mellett döntöttünk.

A Python nyelv - bemutatja Stéfane Fermigier[2].

A Python egy portábilis, dinamikus, bővíthető, ingyenes nyelv, ami lehetővé teszi a programozás moduláris és objektum orientált megközelítését. 1989 óta fejleszti Guido van Rossum és számos önkéntes.

A nyelv jellemzői

Részletezzük egy kicsit a nyelv, pontosabban két jelenlegi implementációjának jellemzőit :

- A Python **portábilis** nemcsak különböző Unix változatokra, hanem *MacOS*, *BeOS*, *NeXTStep*, *MS-DOS* és különböző *Windows* változatokra is. Egy új fordítót írtak Java-ban – Jpython-nak hívják – ami Java *bytekódot* hoz létre.
- Ingyenes, ugyanakkor korlátozás nélkül használható kereskedelmi projektekben.
- Egyaránt megfelel néhányszor tízsoros **scripteknek** és több tízezer soros **komplex projekteknek**.
- **Szintaxisa nagyon egyszerű**, fejlett adattípusokat kombinál (listákat, szótárakat, ...). Nagyon tömör, ugyanakkor jól olvasható programok írhatók vele. Az azonos funkciójú C és C++ (vagy éppen Java) program hosszának gyakran a harmada-ötöde az egyenértékű (bőségesen kommentált és a standard szabályoknak megfelelően prezentált) Python program, ami általában 5-10-szer rövidebb fejlesztési időt és lényegesen egyszerűbb karbantartást jelent.
- A programozó beavatkozása nélkül kezeli az erőforrásokat (memória, filehandlerek, ...) egy **hivatkozás számláló** mechanizmus segítségével (ami hasonlít egy « szemétgyűjtő »-höz (*garbage collector*)), de különbözik attól).
- A Pythonban nincsenek pointerek.
- A Python (opcionálisan) **többszálú** (**multi-thread**).
- **Objektum orientált**. Támogatja a **többszörös öröklést** és az **operátor overloading**-ot. Objektummodelljében – a C++ terminológiát használva – minden metódus virtuális.

A Pythonba – mint a Javaba vagy a C++ újabb verzióiba – egy kivételkezelő rendszer van beépítve, ami lényegesen leegyszerűsíti a hibakezelést.

---

2 Stéfane Fermigier az AFUL (Association Francophone des Utilisateurs de Linux et des logiciels libres = Franciaajkú Linux és szabad software Felhasználók Egyesülete) elnöke. Ez a szöveg a **Programmez!** magazinban 1998 decemberében megjelent cikk kivonata, ami a http://www.linux-center.org/articles/9812/python.html webcímen is elérhető.

- A Python **dinamikus** (az interpreter ki tud értékelni Python kifejezéseket és utasításokat tartalmazó karakterláncokat), **ortogonális** (kevés fogalommal nagyszámú konstrukció írható le), **reflektív** (támogatja a metaprogramozást (például a végrehajtás során képes attribútumokat vagy metódusokat hozzáadni/eltávolítani egy objektumhoz/ból, vagy éppen megváltoztatni az osztályt)) és **introspektív** (számos fejlesztő eszköz - mint a *debugger* és a *profiler* - magában a Pythonban van implementálva).
- A Python dinamikus típusadású, mint a *Scheme* vagy a *SmallTalk*. A programozó által manipulált minden objektumnak a programvégrehajtáskor jól meghatározott típusa van, amit nem kell előre definiálni.
- Jelenleg két implementációja van. Az egyik **interpretált**, melyben a programok portábilis utasításokra vannak lefordítva, majd egy virtuális gép hajtja őket végre (mint a Java esetében, azonban van egy lényeges különbség: mivel a Java statikus típusadású, ezért jóval egyszerűbb egy Java-program végrehajtásának a felgyorsítása, mint egy Python programé). A másik implementáció közvetlenül Java *bytekódot* generál.
- Bővíthető : mint a *Tcl* -t vagy a *Guile* -t, a Pythont könnyen illeszthetjük már meglévő C könyvtárakhoz. Felhasználhatjuk komplex programnyelvek bővítő nyelveként.
- A standard Python könyvtár és a kiegészítő package-ek változatos szolgáltatásokat tesznek hozzáférhetővé : stringek és reguláris kifejezések, standard UNIX szolgáltatások (fileok, pipe-ok, jelek, socketek, szálak, ...), internet protokolok (Web, News, FTP, CGI, HTML...), állandóság (persistence), adatbázisok és grafikus interface-ek.
- A Python **folyamatosan fejlődő** nyelv, ami mögött lelkes felhasználók és fejlesztők közössége áll, akiknek többsége támogatja a szabad szoftvereket. A nyelv alkotója által C-ben írt és karbantartott interpreterrel párhuzamosan egy másik, Javaban írt interpretert is fejlesztenek.
- Végül a Python XML kezelésére alkalmas nyelv.

Több különböző verzió?

Amint említettem, a Pythont folyamatosan fejlesztik. A fejlesztés a célja a termék tökéletesítése. Ezért nem kell azt gondolni, hogy előbb vagy utóbb minden programunkat módosítani kell, hogy azokat egy új, - az előző verziókkal inkompatibilissé vált - verzióhoz adaptáljuk. A könyv példái egymást követően, egy viszonylag hosszú időszak alatt készültek. Egyeseket a Python 1.5.2, míg másokat az 1.6, 2.0, 2.1, 2.2 és végül a 2.3 verziójával írtuk.

Ennek ellenére valamennyi probléma mentesen működik ez utóbbi verzió alatt és bizonyára nagyobb módosítások nélkül fognak működni a következő verziók alatt is.

Telepítse az utolsó verziót és szórakozzon jól !

A Python terjesztése- Bibliográfia

A Python különböző verziói (Windowsra, Unixra, stb.), az eredeti ***oktató anyaga***, a **kézikönyve**, a függvénykönyvtárak ***dokumentációja***, stb. ingyen letölthetők az internetről, a hivatalos Python website-ról : ***http://www.python.org***.

Nagyon jó Pythonnal foglalkozó könyvek kaphatók :

- ***Programming Python***, by Mark Lutz, Second Edition, O'Reilly, March 2001, 1296 pages, ISBN : 0596000855
- ***Learn to program using Python***, by Alan Gauld, Addison-Wesley Professional;, January 15 2001, 288 pages, ISBN**:** 0201709384
- ***Python in a Nutshell,*** by Alex Martelli, 1st Edition, O'Reilly, March 2003, 654 pages, ISBN : 0596001886
- ***Learning Python,*** by Mark Lutz, David Ascher, 2nd Edition, O'Reilly, December 2003, 620 pages, ISBN : 0596002815
- ***Python : How to program***, by Harvey M. Deitel, et al, Prentice Hall; February 4, 2002, 1292 pages, ISBN: 0130923613
- ***Python and Tkinter Programming,*** by John E. Grayson, Manning Publications; 1st edition (January, 2000), 688 pages, ISBN: 1884777813
- **Core Python Programming**, by Wesley J. Chun, Prentice Hall (December 15, 2000), 816 pages, ISBN**:** 0130260363
- **Python Programming On Win32**, by Mark Hammond, Andy Robinson, First Edition January 2000, 669 pages, ISBN: 1565926218
- ***Python Standard Library***, by Fredrik Lundh, O'Reilly, 05/2001, 281 pages, ISBN: 0596000960
- ***Python cookbook***, by Alex Martelli, Anna Martelli Ravenscroft, David Ascher, 2nd Edition, O'Reilly, 03/2005, 807 pages, ISBN: ***0596007973***

A tanárnak, aki oktatási segédletként akarja használni a könyvet

A jegyzettel az a célunk, hogy szélesre tárjuk a kapukat. Tanulmányainknak ezen a szintjén fontosnak tűnik annak bemutatása, hogy a számítógép programozás a fogalmak és módszerek óriási tárháza, melyben mindenki megtalálhatja a neki szimpatikus területet. Nem gondoljuk azt, hogy minden hallgatónak pontosan ugyanazokat a dolgokat kell megtanulni. Inkább azt szeretnénk, hogy mindegyiküknek sikerüljön kifejleszteni egy kicsit eltérő szaktudást, ami lehetővé teszi, hogy mind önmaguk, mind pedig tanulótársaik előtt értékeljék magukat és ha egy nagyobb léptékű projektben való együttműködést ajánlunk nekik, akkor hozzá tudjanak járulni a speciális tudásukkal.

Minden esetre a legfőbb gondunk annak kell lenni, hogy sikerüljön fölkeltenünk az érdeklődést, ami messze a legnagyobb haszon egy olyan nehéz tárgykör esetében mint a számítógép programozás. Nem akarunk úgy tenni, mint akik azt hiszik hogy diákjaik azonnal lelkesedni fognak a szép algoritmusok írása iránt. Inkább arról vagyunk meggyőződve, hogy az érdeklődést csak attól a pillanattól kezdve lehet tartósan fenntartani, amikortól képessé válnak arra, hogy némi önállósággal eredeti, személyes projekt fejlesztéséhez kezdhetnek.

Ezek a meggondolások vezettek minket egy olyan tantárgyszerkezet kialakításához, amit egyesek talán kicsit kaotikusnak fognak találni. A vezérfonal a kitűnő « *How to think like a computer scientist* », amit kicsit kibővítettünk az adatbe-/kivitelre és különösen a *Tkinter* grafikus interface-re vonatkozó elemekkel. Azt szeretnénk, ha diákjaink már programozás tanulmányaik első évének végétől kezdve tudnának kis grafikus alkalmazásokat készíteni.

Egészen konkrétan ez azt jelenti, úgy gondoljuk a tantárgy első évében átfutjuk a jegyzet első nyolc fejezetét. Ez azt tételezi fel, hogy elég gyors ütemben megtárgyalunk egy sor fontos fogalmat (adattípusok, programvégrehajtás vezérlő utasítások, függvények és ciklusok), de nem foglalkozunk azzal túl sokat, hogy a tanulók tökéletesen megértsenek minden egyes fogalmat, mielőtt rátérünk a következőre. Inkább megpróbáljuk rávezetni őket a személyes kutatás és kísérletezés ízére. Sokszor hatékonyabb lesz, ha az egyes fogalmakat és fontos mechanizmusokat adott szituációban, változatos összefüggések között újra elmagyarázzuk.

Elképzelésünk szerint főleg a második év az, amikor megpróbáljuk a megszerzett ismereteket elmélyíteni és struktúrálni. Az algoritmusokat részletesen elemezzük. A projekteket, feladat meghatározásokat és elemző módszereket konzultációkon beszéljük meg. Megköveteljük bizonyos munkák esetén a jegyzőkönyv rendszeres használatát és a technikai jelentések készítését.

A végcél mindegyik tanuló számára egy komoly, eredeti programozási projekt kivitelezése lesz. Ezért törekszünk a fontos fogalmak elméleti tárgyalásának elég korán - a tanév elején - történő befejezéséhez, hogy mindenkinek elég idő álljon a rendelkezésére.

Fontos annak a megértése, hogy jegyzetben közölt számos információ – ami egy sor speciális területet érint (grafikus interface-ek, kommunikáció, adatbázisok kezelése, stb.) - fakultatív tananyag. Ezek csak javaslatok és irányjelzők, amiket azért vettünk be a könyvbe, hogy segítsünk a diákjainknak a tanulmányaikat lezáró személyre szabott projektjük kiválasztásában és megkezdésében. Nem kíséreljük meg egy adott nyelv vagy technikai terület specialistáinak képzését : csak egy kis rálátást szeretnénk nyújtani azokra az óriási lehetőségekre, amik azoknak kínálkoznak, akik veszik a fáradságot és programozói tudásra tesznek szert.

A könyv példái

A könyv példáinak forráskódja letölthető a szerző website-járól :

http://www.ulg.ac.be/cifen/inforef/swi/python.htm

Köszönetnyilvánítás

A jegyzet egy része személyes munka eredménye, míg más – jóval jelentősebb része – információk és jóindulatú tanárok és kutatók által rendelkezésre bocsátott ötletek gyűjteménye. Amint föntebb már jeleztem, az egyik legfontosabb forrásom A.Downey, J.Elkner & C.Meyers : *How to think like a computer scientist* kurzusa volt. Mégegyszer köszönet ezeknek a lelkes tanároknak. Bevallom, Guido van Rossum (a Python főszerzőjének) eredeti oktató anyaga, valamint a Python felhasználók (rendkívül aktív) közösségétől származó példák és különféle dokumentumok is nagymértékben inspiráltak. Sajnos lehetetlen felsorolni valamennyi szöveg hivatkozását, de szeretném elismerésemről biztosítani valamennyiük szerzőit.

Köszönet illeti mindazoknak, akik a Python és kiegészítői, valamint a dokumentáció fejlesztésén dolgoznak, Guido van Rossummal kezve természetesen, de a többieket sem kifelejtve (Szerencsére túlságosan sokan vannak, semhogy valamennyiük nevét fel tudnám itt sorolni).

Köszönet illeti még kollégáimat Freddy Klich-et, Christine Ghiot-t és David Carrera-t a Liège-i St. Jean-Berchmans Intézet tanárait, akik diákjaikkal együtt vállalkoztak rá, hogy belevetik magukat ennek az új kurzusnak a kalandjába és akiknek szintén számos jobbító javasoltuk volt. Külön köszönöm Christophe Morvan-nak az IUT de Marne-la-Vallée tanárának értékes véleményét és bátorítását.

Nagyon köszönöm szerkesztőmnek Florence Leroy-nak az O'Reilly kiadónál, hogy szakértően javította fogalmazási hibáimat és belgicizmusaimat.

Végül megköszönöm feleségem Suzel türelmét és megértését.

A fordító előszava

A Python a leggyorsabban fejlődő nyíltforrású, általános célú objektum orientált programozási nyelv. A programozók 14 %-a használja. Szintaxisa rendkívül egyszerű, könnyen tanulható.

A www.python.org -ról ingyen letölthető és minden korlátozás nélkül szabadon felhasználható, módosítható, terjeszthető. Az interneten rengeteg dokumentáció, példaprogram található, amik segítik a kezdők elindulását.

A gyakorlati élet számos területén alkalmazzák pl.: hálózati alkalmazások, webfejlesztés, numerikus és tudományos alkalmazások, prototipus fejlesztés, gyors alkalmazás fejlesztés (RAD = Rapid Application Development), stb.

Gérard Swinnen könyve pedagógiailag átgondolt, rendkívül logikus bevezetés a Python programozási nyelvbe. Viszonylag kis terjedelme ellenére számos alkalmazási területre ad rálátást a példaprogramok és alapos elemzésük segítségével. A könyv szerves részét képezik a példák, illetve a külön is letölthető működő példaprogramok.

A fordítással az volt a célom, hogy egy iskolai tanításra is alkalmas kiváló könyvet tegyek magyar nyelven hozzáférhetővé az érdeklődőknek. Az olvasó fordítással kapcsolatos észrevételeit köszönettel veszem a python@gportal.hu mail-címen.

Debrecen, 2006. január 10.

Daróczy Péter

1. Fejezet : Programozóként gondolkodni[3]

Mielőtt elkezdjük a programozás tanulást, be kell vezessek néhány fogalmat, melyek ismeretére a továbbiakban szükségünk lesz. Szándékosan egyszerűsítem a magyarázatokat, hogy ne terheljem túl az olvasót. A programozás nem nehéz : elég hozzá egy kis módszeresség és kitartás.

1.1 A programozás

A tantárgy célja, hogy megtanítson programozóként gondolkozni. Ez a gondolkodásmód olyan összetett eljárásokat kombinál, mint amilyeneket a matematikusok, a mérnökök, és a tudósok alkalmaznak.

A programozó a matematikushoz hasonlóan formális nyelveket használ az okfejtések (vagy az algoritmusok) leírására. A mérnökhöz hasonlóan terveket gondol ki, az alkotó részekből szerkezeteket állít össze, és értékeli azok teljesítményét. Mint a tudós megfigyeli az összetett rendszerek viselkedését, magyarázatokat vázol föl, ellenőrzi a jóslatokat.

A programozó fő tevékenysége a problémamegoldás.

Ez különböző képességeket és ismereteket igényel :

- egy problémát különböző módokon kell tudnunk meg/újrafogalmazni,
- innovatív és hatékony megoldásokat kell tudnunk elképzelni,
- ezeket a megoldásokat világosan és komplett módon kell tudnunk kifejezni.

A számítógép programozása lényegében abból áll, hogy részletesen « megmagyarázzuk » egy gépnek - ami nem « érti meg» az emberi nyelvet, csupán karaktersorozatok automatikus kezelésére képes -, hogy mit kell tennie.
A program előre rögzített konvenciók – ezek együttesét programozási nyelvnek nevezzünk - szigorú betartásával kódolt utasítások sorozata. A gép rendelkezik egy eljárással ami úgy dekódolja ezeket az utasításokat, hogy a nyelv minden « szavához » egy pontosan meghatározott akciót rendel.

Az olvasó megtanul programozni, ami már önmagában hasznos, mert fejleszti az intelligenciát. Majd odáig is eljut, hogy a programozást konkrét projektjek megvalósítására használja, amiben biztos örömét fogja lelni.

1.2 Gépi nyelv, programozási nyelv

A számítógép két állapotú elektromos jelek (például egy minimális vagy egy maximális feszültség) sorozatain hajt végre egyszerű műveleteket. Ezek a jelsorozatok egy « minden vagy semmi » típusú logikát követnek. Úgy tekinthetjük őket, mint olyan számok sorozatát, melyek mindíg csak a 0 és az 1 értékeket vehetik fel. Az ilyen számrendszert kettes (bináris) számrendszernek nevezzük.

A számítógép a belső működése során csak bináris számokat tud kezelni. Minden más típusú információt bináris formátumúvá kell átalakítani vagy kódolni. Ez nemcsak a kezelendő adatokra (szövegek, képek, hangok, számok, stb.) igaz, hanem a programokra is, vagyis azokra az utasítás

3 Ennek a fejezetnek a jelentős részét Downey, Elkner és Meyers « How to think like a computer scientist » -jéből fordítottam.

sorozatokra is, amiket azért adunk meg a gépnek, hogy megmondjuk neki, mit kell csinálni az adatokkal.

Az egyetlen « nyelv », amit a számítógép valóban « megért » tényleg nagyon távol van attól amit mi használunk. 1-esek és 0-ák (ezek a bitek) hosszú sorozata, amiket gyakran 8, 16, 32 vagy éppen 64-esével csoportosítva használ. Ez a « gépi nyelv » számunkra érthetetlen. Ahhoz, hogy egy számítógéppel « beszéljünk », olyan fordító rendszereket kell alkalmaznunk, melyek képesek a számunkra érthetőbb kucsszavakat (rendszerint angol szavakat) alkotó karaktersorozatokat bináris számokká alakítani.

Ezeknek a fordító rendszereknek, amiket egy sor konvenció alapján implementálnak, nyilvánvalóan számos változata létezik.

Attól függően, hogy milyen eljárást alkalmaz a fordító : hívjuk ***interpreternek*** vagy ***compilernek*** (lásd lentebb). A ***programozási nyelv*** nagyon pontos szabályokhoz rendelt (önkényesen választott) kulcsszavaknak a készlete. Azt írja le, hogyan rakhatjuk össze ezeket a szavakat olyan « mondatokká », amiket az interpreter vagy a compiler a gép nyelvére (bináris számokra) le tud fordítani.

Az absztrakció szintje alapján beszélhetünk « alacsonyszintű » (pl. : *Assembler*) vagy « magas szintű » nyelvekről (pl.: : *Pascal, Perl, Smalltalk, Clarion, Java*...). Egy alacsonyszintű nyelvet nagyon elemi, nagyon « gépközeli » utasítások alkotják. Egy magas szintű nyelv utasításai absztraktabbak vagy « hatékonyabbak ». Ez azt jelenti, hogy az interpreter vagy a compiler minden utasítást nagyszámú elemi gépi utasításra fordít le.

Az olvasó első programozási nyelvként a **Python**-t fogja megtanulni. Ez egy magas szintű nyelv. A bináris kódra történő fordítása összetett eljárás és mindíg időigényes. Ez kényelmetlennek tűnhet. Valójában a magas szintű nyelveknek rendkívüli előnyeik vannak : egy magas szintű nyelven a programírás ***sokkal egyszerűbb***, jóval kevesebb időbe kerül; annak a valószínűsége, hogy hibákat ejtünk jóval csekélyebb, mintha egy alacsony szintű nyelven programoznánk; a karbantartás (vagyis a későbbi módosítások) és a hibakeresés (debugolás) nagymértékben egyszerűsödnek. Ráadásul egy magas szintű nyelven megírt program gyakran ***hordozható*** (***portable***) lesz, vagyis úgy működtethetjük, hogy nem kell sokat változtatni rajta a különböző gépeken vagy operációs rendszereken. Egy alacsonyszintű nyelven írt program mindíg csak egy géptípuson tud működni. Ahhoz, hogy egy másik géptípuson működjön teljesen át kell írni.

1.3 Compilálás és interpretálás

A programot, ahogyan azt egy szerkesztő programmal (egyfajta speciálizált szövegszerkesztővel) megírjuk mostantól fogva ***forrásprogramnak*** (vagy forráskódnak) nevezzük. Mint már említettem, két fő technika létezik arra, hogy egy ilyen forráskódot a gép által végrehajtható bináris kódra fordítsunk : az interpretáció és a compilatio.

- Az ***interpretáció*** esetén minden egyes alkalommal, amikor végre akarjuk hajtatni a programot, az interpreter programot kell használnunk. Ennél a technikánál a fordító a forrásprogram minden egyes elemzett sorát néhány gépi nyelvű utasításra lefordítja, amiket azonnal végre is hajt. Semmilyen tárgyprogram sem generálódik

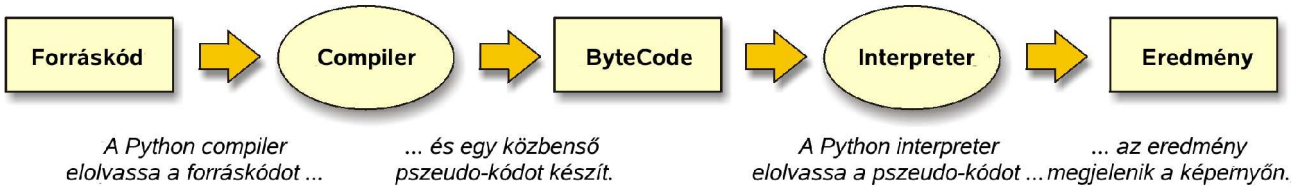
- A ***compilálás*** a teljes forrásszöveg egyszeri lefordításából áll. A fordító program elolvassa a forrásprogram összes sorát és egy új kódot állít elő, amit tárgykódnak (object kód) hívunk. Ez utóbbit mostmár a compilertől függetlenül végrehajtathatjuk és tárolhatjuk egy file-ban (« végrehajtható file »)

Mindegyik technikának vannak előnyei és hátrányai :

Az interpretáció ideális amikor egy nyelvet tanulunk, vagy egy projekttel kísérletezünk. Ezzel a technikával közvetlenül tesztelhető a forráskód minden megváltoztatása anélkül, hogy átmennénk egy compilálási fázison, ami mindíg időigényes.

Ezzel szemben, amikor egy projektnek összetett funkciói vannak, amiket gyorsan kell végrehajtani, a compilatiót részesítjük előnyben. Nyilvánvaló, hogy egy compilált program mindíg lényegesen gyorsabban fog működni, mint az interpretált változata, mivel a számítógépnek a végrehajtás előtt nem kell minden egyes utasítást újra bináris kódra lefordítani.

Egyes modern nyelvek megpróbálják a két módszert kombinálni, hogy mindegyikből a legjobbat hozzák ki. Ez a helyzet a Python és a Java esetében is. Amikor egy forráskódot adunk át a Pythonnak, az először egy gépi kódhoz hasonló közbenső kódot ú.n. ***bytecode***-ot hoz létre, amit aztán egy interpreternek ad át végrehajtásra. A számítógép szempontjából a *bytecode* -ot nagyon egyszerű gépi nyelven interpretálni. Ez az interpretáció tehát sokkal gyorsabb lesz, mint egy forráskód interpretációja.



Ennek a módszernek az előnyei érzékelhetőek :

- Az, hogy állandóan rendelkezünk egy interpreterrel, lehetővé teszi bármilyen kis programrészlet közvetlen tesztelését. Egy alkalmazás bármelyik alkotójának helyes működését igazolhatjuk annak létrehozását követően.
- A lefordított *bytecode* interpretációja nem olyan gyors, mint egy igazi bináris kódé, de számos program számára, beleértve a grafikus programokat is, nagyon megfelel.
- A *bytecode* ***portábilis***. Ahhoz, hogy egy Python vagy egy Java program különböző gépeken végrehajtódjon elég ha mindegyik gépen rendelkezésre áll egy adaptált interpreter.

Az elmondottak egy kicsit bonyolultnak tűnhetnek, az viszont jó hír, hogy mindezt a Python fejlesztőkörnyezet automatikusan végzi el. Elég, ha beírjuk a parancsokat a klaviatúrán, majd <Entert> nyomunk és a Python magára vállalja azok fordítását és interpretálását.

1.4 Programfejlesztés - Hibakeresés (« debug »)

A programozás nagyon összetett és mint minden emberi tevékenység során, ennek során is számos hibát követünk el. Anekdotai okokból a programozási hibákat bug[4]-oknak nevezzük. A detektálásukra és javításukra használt technikák együttesét « *debug* »-nak nevezzük.

Háromféle típusú hiba létezhet egy programban. Illendő megtanulni megkülönböztetni őket:

1.4.1 Szintaxishibák

A Python csak akkor tud végrehajtani egy programot, ha a ***szintaxisa*** teljesen korrekt. Ellenkező esetben a programvégrehajtás leáll és egy hibaüzenetet kapunk. A szintaxis kifejezés azokra a szabályokra utal, melyeket a nyelv szerzői rögzítettek a program struktúrájára vonatkozóan.

Minden nyelvnek van szintaxisa. Például a magyar nyelvben egy mondat mindíg nagy betűvel kezdődik és írásjellel végződik. ezért ez a mondat két szintaxishibát tartalmaz

A közönséges szövegekben néhány kis szintaxishibának nincs jelentősége. Sőt előfordulhat (például a versekben), hogy szándékosan követtek el szintaxishibákat. Ez nem gátolja meg a szöveg megértését.

Ezzel szemben egy számítógép programban a legkisebb szintaxishiba a működés leállását (kiakadás) és egy hibaüzenet kiírását eredményezi. Az olvasó programozói pályafutása első heteiben biztosan sok időt fog eltölteni a szintaxishibái keresésével. Gyakorlattal jóval kevesebb hibát fog elkövetni.

Tartsuk észben, hogy az alkalmazott szavaknak és szimbólumoknak önmagukban semmilyen jelentésük sincs : ezek csak kódsorozatok, amik arra valók, hogy automatikusan bináris számokká legyenek alakítva. Következésként nagyon ügyelnünk kell a nyelv szintaxisának a legaprólékosab betartására.

Szerencse, hogy első programozói lépéseinket egy olyan interpetált nyelvvel tesszük meg, mint a Python. Vele egyszerű és gyors a hibakeresés. Compilált nyelvekkel (mint a C++) minden módosítás után, legyenek azok akármilyen kicsik is, újra kellene fordítani a teljes programot.

1.4.2 Szemantikai hibák

A második hibatípus a ***szemantikai hiba*** vagy logikai hiba. Ha ilyen típusú hiba van valamelyik programunkban, akkor a program tökéletesen hajtódik végre abban az értelemben, hogy semmilyen hibaüzenetet sem kapunk, de az eredmény nem az amit várunk : mást kapunk.

Valójában a program azt teszi, amit mondtunk neki, hogy hajtson végre. A probléma az, hogy amit mondtunk, hogy hajtson végre nem felel meg annak, amit szerettünk volna, hogy a program végrehajtson. A program utasításainak sorrendje nem felel meg a kitűzött célnak. A szemantika (a logika) nem korrekt.

A logikai hibák keresése nehéz feladat lehet. Elemezni kell az output-ot és meg kell próbálni egymás után reprodukálni azokat a műveleteket, amiket a gép az egyes utasítások után végrehajtott.

4 A "*bug*" angol eredetű kifejezés, olyan kis, kellemetlen rovarokat jelent, mint amilyenek a poloskák. Az első számítógépek elektroncsövei meglehetősen nagy feszültséget igényeltek. Számos alkalommal megtörtént, hogy ezek a rovarok bemásztak az áramkörök közé és áramütést kaptak, szénné égett testük rövidzárakat és így érthetetlen meghibásodásokat okozott.

1.4.3 Végrehajtás közben fellépő hibák

A hibák harmadik típusát a végrehajtás közben fellépő hibák (*Run-time error*) képezik. Ezek csak akkor lépnek fel, amikor a program működése során speciális körülmények állnak elő (például a program egy már nemlétező file-t kísérel meg olvasni). Ezeket a hibákat ***kivételeknek*** (***exception)*** is hívják, mert ezek általában jelzik, hogy valami kivételes történt (amit nem láttunk előre). Többször fogunk találkozni ilyen típusú hibával amikor egyre nagyobb méretű projekteket fogunk programozni.

1.5 Hibakeresés és kísérletezés

A tanulás során elsajátítandó egyik legfontosabb készség a hatékony programhiba keresés (debugolás). Ez néha idegesítő, de mindíg nagyon sokrétű szellemi tevékenység, aminek a során éleslátásról kell tanubizonyságot tenni.

Emlékeztet egy rendőrségi kihallgatásra. Megvizsgáljuk a tényeket és magyarázó feltevéseket kell tennünk, hogy rekonstruáljuk a tapasztalt eredményeket adó folyamatokat és eseményeket.

Hasonlít a tudományos kísérletekre. Van egy elképzelésünk arról, hogy mi nem működik, módosítjuk a programunkat és újra kipróbáljuk. Kialakítottunk egy feltételezést, ami lehetővé teszi annak megjóslását, hogy mit kell eredményezzen a módosítás. Ha a jóslat beigazolódik, tettünk egy lépést egy müködő program irányába. Ha a jóslatunk tévesnek bizonyul, akkor egy újabb hipotézist kell alkotnunk. Ahogyan azt Sherlock Holmes mondja : « Amikor kizártuk a lehetetlent, annak kell lennie az igazságnak, ami megmarad, még ha az valószínűtlen is *» (*A. Conan Doyle*, A négyes jel).*

Egyeseknek a « programozás » és « debugolás » pontosan ugyanazt a dolgot jelenti.
Ezzel azt akarják mondani, hogy a programozás ugyanannak a programnak az állanó módosításából és javításából áll, mindaddig, míg végül a program úgy viselkedik, ahogyan akarjuk, hogy viselkedjen. Az elképzelés az, hogy a programkészítés mindíg egy már működő (azaz debugolt) vázzal kezdődik, amihez rétegről rétegre kis módosításokat teszünk hozzá, fokozatosan javítjuk a hibákat, hogy végül a folyamat mindegyik szakaszában egy működő programunk legyen.

Például tudjuk, hogy a Linux egy több ezer kódsorból álló operációs rendszer (tehát egy nagy program). Ennek ellenére kezdetben úgy indult mint egy kicsi, egyszerű program, amit Linus Torvalds arra fejlesztett ki, hogy tesztelje az Intel 80386 processzor sajátosságait. Larry GreenField szerint (« *The Linux user's guide* », béta verzió 1) : « *Linus első programjainak egyike egy kis program volt, aminek a feladata az AAAA karakterlánc BBBB karaklánccá való alakítása volt. Ez az, ami később a Linux-szá vált ! »*.

Az előzőek nem jelentik azt, hogy homályos elképzelésből kiindulva szukcesszíven approximálva akarnánk programozni. Amikor egy jelentős progamprojektbe kezdünk, minden erőnkkel arra kell törekednünk, hogy a lehető legjobb ***részletes feladat meghatározást*** írjuk meg, ami az elképzelt alkalmazás tervén fog alapulni.

Különböző módszerek léteznek erre a ***problémaelemzésre***, azonban ezek tanulmányozása meghaladja ennek a jegyzetnek a kereteit. További információkért és hivatkozásokért forduljon az olvasó a tanárához.

1.6 Természetes és formális nyelvek

Azok a természetes nyelvek, amiket kommunikációra használnak az emberek. Nem konstruált nyelvek (még akkor sem, ha egyesekben megpróbálnak rendszert teremteni) : természetes módon fejlődnek.

A ***formális nyelveket*** speciális alkalmazásokat szem előtt tartva fejlesztették ki. Így például a matematikusok által használt jelölésrendszer egy olyan formális nyelv, ami különösen hatékonyan fejezi ki a különböző számok és méretek közötti relációkat. A vegyészek egy formális nyelvet használnak a molekulák szerkezetének bemutatására, stb.

> ***A programozási nyelvek azok a formális nyelvek, amiket algoritmusok leírására fejlesztettek ki.***

Mint már föntebb jeleztem, a formális nyelvek szintaxisa rendkívül szigorú szabályoknak engedelmeskedik. Például a `3+3=6` egy matematikailag korrekt ábrázolásmód, míg a `$3=+6` nem az. Ugyanígy a $\mathbf{H_2O}$ kémiai képlet korrekt, de a $\mathbf{Zq_3G_2}$ nem.

A szintaxisszabályokat nem csak a nyelv szimbólumaira (például a Zq kémiai szimbólum nem megengedett, mert semmilyen elemnek sem felel meg), hanem azok kombinálási módjára is alkalmazzuk. Ezért, jóllehet a `6+=+/5-` matematikai egyenlet csak megengedett szimbólumokat tartalmaz, azonban azok inkorrekt elrendezése semmit sem jelent.

Egy mondat olvasásakor el kell jutnunk odáig, hogy elképzeljük a mondat logikai szerkezetét (még akkor is ha ezt az esetek többségében nem tudatosan tesszük). Például amikor « *A pénzdarab leesett.* » mondatot mondjuk, megértjük, hogy « A *pénzdarab* » az alany és a « leesett » az állítmány. Az elemzés lehetővé teszi, hogy megértsük a mondat jelentését, logikáját (szemantikáját). Analóg módon a Python interpreternek elemezni kell a forrásprogramunk szerkezetét, hogy a jelentést kihámozza belőle.

A természetes és a formális nyelveknek sok közös jellemzője van (szimbólumok, szintaxis, szemantika), de nagyon jelentős különbségeket is vannak közöttük :

Kétértelműség.

A természetes nyelvek tele vannak kétértelműségekkel, amik az esetek többségében a szövegkörnyezet segítségével megszüntethetők. Például eltérő jelentést tulajdonítunk az ár szónak egy termékről, illetve egy árvízről szóló szövegben. Egy formális nyelv nem tartalmazhat kétértelműségeket. Minden utasításnak egyetlen, szövegkörnyezettől független jelentése van.

Redundancia.

A természetes nyelvekben sok a redundancia azért, hogy ezeket a kétértelműségeket és az információ átvitelbeli számos hibát és veszteséget kompenzálják (mondatainkban többször különböző formában megismételjük ugyanazt, hogy biztosak legyünk benne, hogy megérttettük magunkat). A formális nyelvek sokkal tömörebbek.

Irodalmiság.

Az irodalmi szövegek tele vannak képekkel és metaforákkal. Egy formális nyelvben ezzel szemben a kifejezéseket szó szerint kell venni. Ha egy bizonyos szövegkörnyezetben azt mondom, hogy « leesett a kétforintos », lehetséges, hogy szó sincs igazi kétforintosról, sem pedig leesésről. Ezzel szemben egy formális nyelvben a kifejezéseket szó szerint kell érteni.

A természetes nyelvek használatához szokva sokszor nehezen alkalmazkodunk a formális nyelvek szigorához. Ez az egyik nehézség, amin túl kell jutni, hogy eljussunk egy hatékony program-analitikus gondolkodásmódjáig.

A jobb megértés érdekében hasonlítsunk össze még néhány szövegtípust :

Vers :

A versekben a szavakat mind zeneiségük, mind jelentésük miatt használják és a keresett hatás főként érzelmi. Tobzódnak a metaforákban és kétértelműségekben.

Prózai szöveg :

A szavak szószerinti jelentése fontosabb bennük, és a mondatok úgy vannak struktúrálva, hogy megszüntessék a kétértelműségeket, de ez soha sem sikerül teljesen. Gyakran szükségesek a redundanciák.

Számítógépprogram :

A szöveg jelentése egyértelmű és szószerinti. Tökéletesen megérthető csak a szimbólumok és a szerkezet elemzése révén. Tehát automatizálni lehet az elemzését.

Néhány javaslat, hogy hogyan olvassunk számítógépprogramot (vagy bármilyen formális nyelven írt szöveget).

Tartsuk észben, hogy a formális nyelvek sokkal tömörebbek, mint a természetes nyelvek, ami azt jelenti, hogy az olvasásukhoz több időre van szükség. Ráadásul a struktúra ezeknél nagyon fontos. Általában az sem jó ötlet, ha egy menetben az elejétől a végéig olvasunk el egy programot. Ehelyett gyakoroljuk a programelemzést fejben a szimbólumok azonosításával és a szerkezet értelmezésével.

Végül emlékezzünk rá, hogy minden részletnek jelentősége van. Különösen figyelnünk kell a kis és nagybetűkre valamint az írásjelek használatára. Minden ilyen hiba (még a látszólag legkisebb is, mint egy vessző kifelejtése) jelentősen módosíthatja a kód jelentését és így a program lefutását.

2. Fejezet : Az első lépések

Ideje munkához látnunk. Pontosabban, megkérjük a számítógépet, hogy dolgozzon helyettünk. Pédául azt az parancsot adjuk neki, hogy végezzen el egy összeadást és írja ki az eredményt.

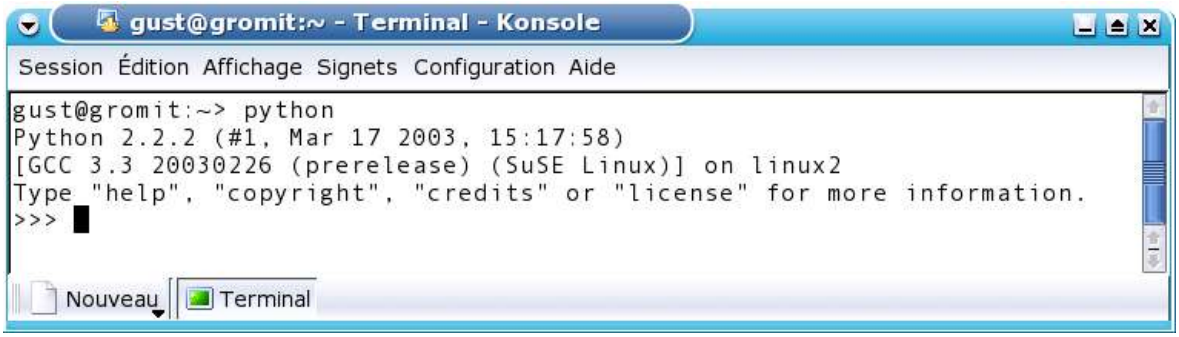
Ehhez « utasításokat » és adatokat kell megadnunk, melyekre alkalmazni akarjuk az előbbieket.

2.1 Számolás a Pythonnal

A Pythonnak az a sajátossága, hogy többféle módon használhatjuk.
Először ***interaktiv módban***, fogjuk használni, vagyis úgy, hogy a billentyűzet segítségével közvetlenül folytathatunk vele párbeszédet. Így igen gyorsan felfedezhetjük a nyelv számos lehetőségét. Később megtanuljuk, hogyan kell létrehozni és diszkre elmenteni programokat (scripteket).

Az interpretert közvetlenül a parancssorból indíthatjuk (egy Linux « shell »-ben, vagy *Windows* alatt egy *DOS* ablakból) : elég, ha begépeljük a **"python"** parancsot (feltéve, hogy a program fel van telepítve)).

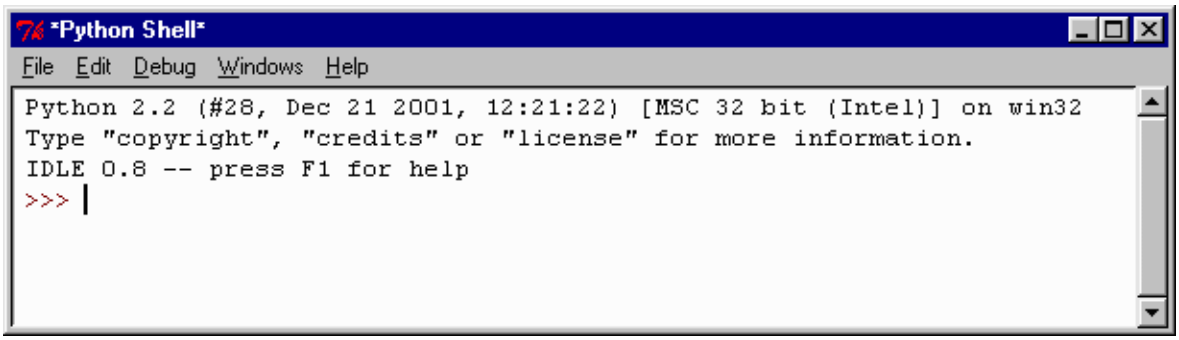
Ha grafikus interface-t használunk, mint a *Windows*, *Gnome, WindowMaker* vagy *KDE*, valószínűleg inkább egy « terminálablakban » fogunk dolgozni, vagy egy specializált fejlesztő környezetben, mint amilyen az ***IDLE***. Terminálablakban (Linux[5] alatt) a következő jelenik meg :



5 *Windows* alatt választhatunk a Guido Van Rossum által írt *IDLE* fejlesztőkörnyezet, amit magam előnyben részesítek, és a Mark Hammond által fejlesztett *PythonWin* között. Léteznek más kifinomult fejlesztőkörnyezetek is, mint a kiváló *Boa Constructor* (ez hasonlóan működik, mint a *Delphi*), de úgy vélem, hogy ez nem való kezdőknek. További felvilágosításért a Python websiteot érdemes megnézni.

Linux alatt inkább a *WindowMaker* grafikus környezetet részesítem előnyben (mint a *KDE*-t vagy a *Gnome*-ot, amik túl sok erőforrást vesznek igénybe). Megnyitunk egy egyszerű terminálablakot a Python interpreter indításához vagy a scriptek végrehajtásához és a kitűnő Nedit programot hívjuk, ami a scriptek szerkesztésére való.

Windows alatt az *IDLE* fejlesztő környezet kinézete az alábbi képre hasonlít :



A >>> jel a főprompt, ami azt jelzi, hogy az interpreter készen áll egy utasítás végrehajtására.

Például az interpretert azonnal egyszerű irodai kalkulátorként használhatjuk. Ellenőrizzük az alábbi parancsokat (Váljon szokásunkká, hogy egy füzetbe felírjuk a képernyőn megjelenő eredményeket) :

```
>>> 5+3

>>> 2 - 9          # a szóközök opcionálisak

>>> 7 + 3 * 4           # a matematikai operátorok hierarchiája
                        # megmarad-e ?
>>> (7+3)*4

>>> 20 / 3              # meglepetés !!!
```

Megállapíthatjuk, hogy a +, -, * és /. aritmetikai operátorok összeadás-, kivonás-, szorzás- és osztásra valók. A zárójelek működnek.

Alapértelmezésben viszont az osztás *egészosztás*, ami azt jelenti, hogy ha egész argumentumokat adunk meg, akkor az eredmény egy (csonkolt) egész lesz, mint a fenti utolsó példában. Ha azt akarjuk, hogy a Python az argumentumát valós számként értelmezze, akkor ezt úgy tudatjuk vele, hogy a számba egy tizedespont teszünk[6].

Próbáljuk ki például :

```
>>> 20.0 / 3            # (hasonlítsuk össze az eredményt az előző példáéval)

>>> 8./5
```

Ha egy műveletet vegyes típusú argumentumokkal hajtunk végre (egészekkel és valósokkal), akkor a Python az operandusokat automatikusan valós típusúvá alakítja át mielőtt elvégzi a műveletet. Próbáljuk ki :

```
>>> 4 * 2.5 / 3.3
```

6 Valamennyi programozási nyelvben : a tizedes szeparátor mindíg egy pont. Az informatikában a valós számokat gyakran lebegőpontos számoknak, vagy ***float***-típusúaknak nevezik.

2.2 Adatok és változók

A későbbiekben lesz alkalmunk részletezni a különböző numerikus adattípusokat, de előtte egy nagyon fontos fogalomról beszélünk:

Egy számítógépprogram lényegét az ***adat***manipulációk képezik. Ezek az adatok nagyon különbözőek lehetnek (lényegében minden, ami *digitalizálható*[7]), de a számítógép memóriájában végleg ***bináris számok véges sorozatává*** egyszerűsödnek.

Ahhoz, hogy a program (bármilyen nyelven is legyen megírva) hozzá tudjon férni az adatokhoz, nagyszámú különböző típusú ***változót*** használ.

Egy programozási nyelvben egy változó majdnem mindegy milyen változónévként jelenik meg, a számítógép számára egy ***memóriacím***et jelölő ***hivatkozásról*** van szó, vagyis egy meghatározott helyről a RAM-ban.

Ezen a helyen egy jól meghatározott ***érték*** van tárolva. Ez az igazi adat, ami bináris számok sorozata formájában van tárolva, de ami az alkalmazott programozási nyelv szemében nem feltétlenül egy szám. Szinte bármilyen « objektum » lehet, ami elhelyezhető a számítógép memóriájában, mint például: egy egész, egy valós, egy komplex szám, egy vektor, egy karakterlánc, egy táblázat, egy függvény, stb.

A programozási nyelv különböző ***változótípusokat*** (*egész, valós, karakterlánc, lista, stb.*) használ a különböző lehetséges tartalmak egymástól történő megkülönböztetésére. A következő oldalakon ezeket fogom elmagyarázni.

7 *Igazán mit digitalizálhatunk ?* Ez egy rendkívül fontos kérdés, amit az általános számítástechnika kurzuson kell megtárgyalnunk.

2.3 Változónevek és foglalt szavak

A változóneveket mi választjuk meg meglehetősen szabadon. Ennek ellenére törekednünk kell, hogy jól válasszuk meg őket. Előnyben kell részesíteni az elég rövid, amennyire lehetséges explicit neveket, amik világosan kifejezik, hogy mit tartalmaz az illető változó. Például: az olyan változónevek, mint ***magasság, magas***, vagy ***mag*** jobbak a magasság kifejezésére, mint **x**.

Egy jó programozónak ügyelni kell arra, hogy az utasítássorait könnyű legyen olvasni.

Ezen kívül a Pythonban a változóneveknek néhány egyszerű szabálynak kell eleget tenni :

- A változónév az (a → z , A → Z) betűk és a (0 → 9) számok sorozata, aminek mindíg betűvel kell kezdődni.
- Csak az ékezetnélküli betűk a megengedettek. A szóközök, a speciális karakterek, mint pl.: $, #, @, stb. nem használhatók. Kivétel a _ (aláhúzás).
- *A kis- és nagybetűk különbözőnek számítanak.*
  ***Figyelem** : Jozsef, jozsef, JOZSEF különböző változók. Ügyeljünk erre !*

Váljon szokásunkká, hogy a változóneveket kisbetűkkel írjuk (a kezdő betűt is[8]) . Egy egyszerű konvencióról van szó, de ezt széles körben betartják. Nagybetűket csak magának a szónak a belsejében használjunk, hogy fokozzuk az olvashatóságot, mint például: tartalomJegyzék.

További szabály: nem használható változónévként az alább felsorolt 28 «foglalt szó » (ezeket a Python használja) :

and	**assert**	**break**	**class**	**continue**	**def**
del	**elif**	**else**	**except**	**exec**	**finally**
for	**from**	**global**	**if**	**import**	**in**
is	**lambda**	**not**	**or**	**pass**	**print**
raise	**return**	**try**	**while**	**yield**	

8 A nagybetűkkel kezdődő szavak nincsenek tiltva, de ezeket inkább az osztályokat (***class***-okat) jelölő változóknak tartjuk fenn (az osztály fogalmát később fogom elmagyarázni).

2.4 Hozzárendelés (vagy értékadás)

Mostmár tudjuk, hogy hogyan válasszuk meg jól egy változónak a nevét. Lássuk most, hogyan definiálhatunk egy változót és hogyan rendelhetünk hozzá ***értéket***. A változóhoz való « érték hozzárendelés » vagy « értékadás » kifejezések egyenértékűek. Egy olyan műveletet jelölnek, amely kapcsolatot teremt a változónév és a változó értéke (tartalma) között.

A Pythonban számos más nyelvhez hasonlóan a hozzárendelés műveletét az ***egyenlőségjel*** reprezentálja[9] :

```
>>> n = 7                    # n-nek a 7-et adjuk értékül
>>> msg = "Mi újság ?"            # A "Mi újság ?" értéket adjuk msg-nek
>>> pi = 3.14159                  # pi nevű változóhoz hozzárendeljük az értékét
```

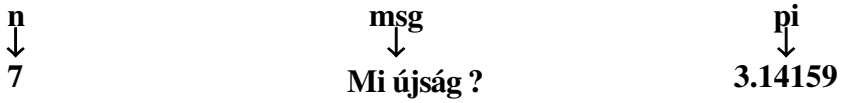
A fenti pythonos értékadó műveletek klasszikusak. Miután ezeket végrehajtottuk, a számítógép memóriájának különbözö helyein:

- három változónév van: **n**, **msg** és **pi**
- három bytesorozat van, melyekben a **7** egész szám, a **Mi újság ?** karakterlánc és a **3,14159** valós szám van kódolva.

A fenti három értékadó utasítás mindegyikének a hatása több művelet végrehajtása a számítógép memóriájában: :

- változónév létrehozása és bejegyzése a memóriába ;
- változónévhez jólmeghatározott típus rendelése (ezt a következő oldalon fogom magyarázni) ;
- egy speciális érték létrehozása és tárolása a memóriában ;
- kapcsolat létesítése a változónév és a megfelelő érték memóriahelye között (mutatók belső rendszere segítségével)

Az elmondottakat az alábbi **állapotdiagrammal** szemléltethetjük :



A három változónév a memória egy speciális részében az ú.n. **névtérben** (**namespace**) tárolt hivatkozás, míg a megfelelő értékek másutt, egymástól gyakran távol helyezkednek el. A későbbiekben alkalmunk lesz pontosítani ezt a koncepciót.

9 Fontos, hogy megértsük, hogy itt semmiféle egyenlőségről sincs szó. Egy más szimbolizmust is választhattunk volna, mint például **n** ← **7** , ahogyan ezt gyakran megteszik más, algoritmusokat leíró pszeudonyelvekben azért, hogy jelezzék, egy tartalomnak (a 7 értéknek) egy konténerhez (a változóhoz) való hozzárendeléséről van szó.

2.5 Változó értékének a kiírása

A fenti gyakorlat következményeként három változónk van: **n, msg, pi**
Két lehetőség van az értékük képernyőre történő kiíratására. Az első: a billentyűzeten beírjuk a változó nevét, majd <Enter>. A Python válaszul kiírja a megfelelő értéket:

```
>>> n
7
>>> msg
"Mi újság ?"
>>> pi
3.14159
```

Az interpreter egy másodlagos funkcionalitásáról van itt szó, aminek az a célja, hogy megkönnyítse az életünket, amikor egyszerű parancssor gyakorlatokat végzünk. Programban mindíg a **print** utasítást használjuk:

```
>>> print msg
Mi újság ?
```

Vegyük észre a kétféle kiíratási móddal kapott eredmény közötti apró különbséget. A **print** utasítás szigorúan csak a változó értékét írja ki, úgy ahogyan az kódolva volt, míg a másik módszer (az, amelyik csak a változónév beírásából áll) az idézőjeleket is (azért, hogy emlékeztessen a változó típusára : erre még vissza fogok térni).

2.6 A változók tipusadása

A Pythonban nem szükséges a változók használata előtt a típusuk definiálása érdekében speciális programsorokat írni. Elég ha hozzárendelünk egy értéket egy változónévhez. A Python a változót ***automatikusan azzal a típussal hozza létre, ami a legjobban megfelel az értéknek***. Az előző gyakorlatban például az **n**, **msg** és **pi** változók mindegyikét automatikusan hozta létre különböző típusokkal (rendre « egész szám », « karakterlánc », « lebegőpontos szám » típusokkal).

Ez egy érdekes sajátossága a Pythonnak, ami alapján a programnyelveknek ahhoz a csoportjához soroljuk, amelyikhez a *Lisp*, *Scheme* és néhány más nyelv tartozik. A Python ***dinamikus típusadású nyelv***, szemben a ***statikus típusadású*** nyelvekkel mint amilyenek a C++ vagy a Java. Ezekben a nyelvekben először mindíg speciális utasításokkal deklarálni (definiálni) kell a változók nevét és típusát és csak ez után lehet tartalmat rendelni hozzájuk, aminek természetesen kompatibilisnek kell lenni a deklarált típussal.

A statikus típusadást a compilált nyelvek részesítik előnyben, mert az lehetővé teszi a fordítás (aminek a kódja egy « befagyott » bináris kód) optimalizálását.

A dinamikus típusadás a magasszintű logikai konstrukciók (metaprogramozás, reflexivitás) leírását teszi könnyebbé, különösen az objektum orientált programozás (polimorfizmus) vonatkozásában. Egyszerűsíti az olyan adatstruktúrák használatát, mint amilyenek a listák és a szótárak (dictionary).

2.7 Többszörös értékadás

A Pythonban egyszerre több változóhoz rendelhetünk értékeket. Például :

```
>>> x = y = 7
>>> x
7
>>> y
7
```

Egyetlen operátor segítségével ***párhuzamosan*** is végezhetünk ***értékadást***:

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Ebben a példában az **a** és **b** egyidejűleg veszi fel a 4 és 8.33 értékeket.

*Figyelem : a tizedes jegyeket **tizedes ponttal** választjuk el az egészrésztől, ahogy az az angol nyelvterületen szokásos. A **vesszőt** a különböző elemek (argumentumok, stb.) elválasztására használjuk ahogyan a példában láttuk, mind maguknak a változóknak, mind pedig a hozzájuk rendelt értékeknek az elválasztására.*

(2) Gyakorlatok

2.1. Írja le a lehető legérthetőbben és teljesebben, hogy mi történik az alábbi példa három sorában :

```
>>> szelesseg = 20
>>> magassag= 5 * 9.3
>>> szelesseg * magassag
930
```

2.2. Rendelje hozzá a 3, 5, 7 értékeket az a, b, c változókhoz.
Végezze el az a - b/c műveletet. Matematikailag korrekt-e az eredmény ?
Ha nem az, hogyan kell eljárni hogy az legyen ?

2.8 Operátorok és kifejezések

Az értékeket és a rájuk hivatkozó változókat ***operátorok*** segítségével ***kifejezésekké*** kombináljuk. Példa :

```
a, b = 7.3, 12
y = 3*a + b/5
```

Ebben a példában az a és b változókhoz először hozzárendeljük a **7.3** és **12** értékeket.
A Python, ahogy azt az előzőekben magyaráztam, automatikusan a « valós » típust rendeli az **a** és az « egész » típust a **b** változóhoz.

A példa második sorában az új **y** változóhoz rendeljük hozzá egy kifejezés eredményét, ami a * , + és / **operátorokat** kombinálja az **a, b, 3** és **5 operandusokkal**. Az operátorok speciális szimbólumok, amiket olyan egyszerű matematikai műveletek reprezentálására használunk, mint az összeadás vagy a szorzás. Az operandusok az értékek, amiket az operátorok segítségével kombinálunk.

A Python minden egyes kifejezést kiértékel, amit beírunk neki, legyen az akármilyen bonyolult és ennek a kiértékelésnek az eredménye mindíg egy érték. Ehhez az értékhez automatikusan hozzárendel egy típust, ami függ attól, hogy mi van a kifejezésben. A fenti példában **y** valós típusú lesz, mert a kiértékelt kifejezés legalább egy valós változót tartalmaz.

Nemcsak a négy matematikai alapművelet operátora tartozik a Python operátoraihoz. Hozzájuk kell venni a hatványozás operátorát **, néhény logikai operátort, a karakterláncokon működő operátorokat, az azonosságot és tartalmazást tesztelő operátorokat, stb. Minderről a későbbiekben újra fogok beszélni.

Rendelkezésünkre áll a ***modulo*** operátor, amit a **%** szimbólum jelöl.
Ez az operátor egy számnak egy másik számmal való ***egészosztásából származó maradékát*** adja meg. Próbáljuk ki :

```
>>> 10 % 3              (jegyezzük meg, hogy mi történik !)
>>> 10 % 5
```

A későbbiekben nagyon hasznos lesz, amikor azt vizsgáljuk, hogy egy **a** szám osztható-e egy **b** számmal. Elég azt megnézni, hogy az **a % b** eredménye egyenlő e nullával.

Gyakorlat :

2.3. Ellenőrizze a következő utasítássorokat. Írja le a füzetébe, hogy mi történik :

```
>>>  r , pi = 12, 3.14159
>>>  s = pi * r**2
>>>  print s
>>>  print type(r), type(pi), type(s)
>>>
```

Véleménye szerint mi a haszna a **type()** *függvény*nek ?
(Megjegyzés : a *függvényeket* a későbbiekben részletesen be fogom mutatni).

2.9 A műveletek prioritása

Ha egy kifejezésben egynél több operátor van, akkor a műveletek végrehajtásának sorrendje a ***prioritási szabályoktól*** függ. A Pythonban a prioritási szabályok ugyanazok, mint amiket matematikából tanultunk :

- ***zárójelek***. Ezek a legmagasabb prioritásúak. Azt teszik lehetővé, hogy az általunk kívánt sorrendben történjen egy kifejezés kiértékelése.
  Így `2*(3-1) = 4` , és `(1+1)**(5-2) = 8`.
- ***hatványozás***. A hatványok kiszámolására a többi művelet előtt kerül sor.
  Így `2**1+1 = 3` (és nem 4), és `3*1**10 = 3` (és nem 59049 !).
- ***szorzás*** és ***osztás***, azonos prioritással. Ezeket az ***összeadás*** és ***kivonás*** előtt értékeli ki.
  Így `2*3-1 = 5` (és nem 4), és `2/3-1 = -1` (Emlékezzünk vissza, hogy alapértelmezésben a Python ***egészosztást*** végez.)
- Ha két operátor azonos prioritású, akkor a végrehajtásuk balról jobbra történik.
  Így az `59*100/60` kifejezésben először a szorzást, majd ezt követően , az `5900/60` osztást végzi el, aminek az eredményet `98`. Ha először az osztást végezné el, akkor az eredmény `59` lenne (emlékezzünk rá, hogy itt egy egészosztásról lenne szó).

2.10 Kompozíció

Eddig egy programozási nyelv különböző elemeit tanulmányoztuk : a ***változókat***, a ***kifejezéseket*** és az ***utasításokat,*** de azt nem vizsgáltuk, hogy hogyan lehet ezeket egymással kombinálni.

Egy magasszintű programozási nyelv egyik erőssége, hogy különböző elemek kombinálásával összetett utasításokat hozhatunk létre. Így például, ha tudjuk hogyan adunk össze két számot és hogyan íratunk ki egy értéket, akkor a két utasítást egyetlen utasítássá kombinálhatjuk :

```
>>> print 17 + 3
>>> 20
```

Ez a nyilvánvalónak tűnő tulajdonság fogja lehetővé tenni az összetett algoritmusok világos és tömör programozását. Példa :

```
>>> h, m, s = 15, 27, 34
>>> print "az éjfél óta eltelt másodpercek száma = ", h*3600 + m*60 + s
```

Figyelem : van egy megszorítás arra nézve, hogy miket lehet kombinálni :

Amit egy kifejezésben az egyenlőségjel baloldalán helyezünk el, annak mindíg egy változónak kell lenni, nem pedig egy kifejezésnek. Ez abból a tényből fakad, hogy az egyenlőségjelnek nem ugyanaz a jelentése, mint a matematikában. Mint már említettem, egy értékadási művelet jeléről van szó (egy bizonyos tartalmat teszünk egy változóba), nem pedig egy egyenlőségjelről. Az egyenlőségjelet (például egy feltétel ellenőrzésében) egy kicsit később fogom bemutatni.

Így például az **m + 1 = b** utasítás ***szabálytalan***.

A matematikában viszont elfogadhatatlan **a = a + 1** -et írni, pedig ez az írásmód igen gyakori a programozásban. Az **a = a + 1** utasítás az **a** változó értékének eggyel történő megnövelését (másként: incrementálását) jelenti.

Hamarosan alkalmunk lesz visszatérni erre. Előtte azonban egy másik nagyon fontos fogalmat kell tisztáznunk.

3. Fejezet : Az utasításfolyam vezérlése

Az első fejezetben láttuk, hogy az elemző programozó alaptevékenysége a problémamegoldás. Egy informatikai probléma megoldásához a ***tevékenységeket*** mindíg bizonyos ***sorrendben*** kell végezni. A megfelelő struktúrában és sorrendben leírt tevékenységeket ***algoritmusnak*** nevezzük.

A tevékenységek végrehajtási sorrendjét meghatározó utasításcsoportok a ***vezérlő struktúrák***. A modern programozásban csak három van belőlük : a ***szekvencia*** és a ***kiválasztás***, melyeket ebben a fejezetben írok le, és a következő fejezetben tárgyalt ***ismétlés***.

3.1 Utasítás szekvencia[10]

Ha nem mondunk mást, akkor egy program utasításai egymás után abban a sorrendben lesznek végrehajtva, ahogyan a scriptben le voltak írva.

Azt az « útvonalat », amit a Python a programvégrehajtás során követ ***utasításfolyam***nak nevezzük. Az ezt módosító utasítások az ***utasításfolyam vezérlő utasítások***.

Normálisan a Python az utasításokat az elsőtől az utolsóig végrehajtja, kivéve ha egy ***feltételes utasítással*** találkozik, mint amilyen az **if** utasítás, amit ez után fogok leírni (a programhurkok kapcsán fogunk vele találkozni). Egy ilyen utasítás lehetővé fogja tenni a programnak, hogy a körülményektől függően más végrehajtási utakat kövessen..

3.2 Kiválasztás vagy feltételes végrehajtás

Ha igazán hasznos programokat szeretnénk írni, akkor olyan technikákra van szükség, amik lehetővé teszik a programvégrehajtás különböző irányokba történő átirányítását attól függően, hogy milyen feltételekkel találkozunk. Ehhez olyan utasítások kellenek, amikkel ***tesztelni lehet egy bizonyos feltételt*** és következményként módosítani lehet a program viselkedését.

Az **if** utasítás a legegyszerűbb ezek közül a feltételes utasítások közül. Kísérlet képpen írjuk be a Python editorába a következő két sort :

```
>>> a = 150
>>> if (a > 100):
...
```

Az első utasítás az **a** változóhoz hozzárendeli a 150 értéket. Eddig semmi új nincs.
Amikor viszont beírjuk a második sort, azt tapasztaljuk, hogy a Python újfajta módon viselkedik. Ha nem felejtettük le a « : » karaktert a sor végéről, akkor megállapíthatjuk, hogy a *főpromptot* (>>>) most egy *másodlagos prompt helyettesíti,* ami három pontból áll[11].

Ha az editorunk nem teszi meg automatikusan, akkor most tabulálnunk kell (vagy 4 betűközt kell írnunk) mielőtt beírjuk a következő sort, úgy hogy az az előző sorhoz képpest beljebb legyen (vagyis be legyen húzva). Képernyőnkön a következőknek kell lenni :

```
>>> a = 150
>>> if (a > 100):
...     print "a meghaladja a százat"
...
```

10 Jelen esetben a ***szekvencia*** kifejezés egymást követő utasítások sorozatát jelöli. Ebben a könyvben ezt a elnevezést egy Python fogalom számára tartjuk fönn, ami a magába foglalja a ***karakterkáncokat***, ***tuple-ket*** és ***listákat*** (lásd a későbbiekben).

11 A Python windowsos editorának bizonyos verzióiban a másodlagos prompt nem jelenik meg.

Nyomjunk még egy <Enter> -t. A program végrehajtódik és a következőt kapjuk :

```
a meghaladja a százat
```

Kezdjük újra a gyakorlatot, de most **a = 20** -szal az első sorban : most a Python semmit sem ír ki.

Azt a kifejezést, amit zárójelek közé tettünk, mostantól fogva feltételnek nevezzük. Az **if** ennek a feltételnek a tesztelését teszi lehetővé. Ha a feltétel igaz, akkor a « : » után beljebb igazított utasítást hajtja végre a Python. Ha a feltétel hamis, semmi sem történik. Jegyezzük meg, hogy az itt alkalmazott zárójelek opcionálisak a Pythonban. Ezeket az olvashatóság javítása érdekében alkalmazam. Más nyelvekben kötelezők lehetnek.

Kezdjük újra, írjunk még két sort az előzőekhez, úgy ahogyan azt alább látjuk. Ügyeljünk rá, hogy a negyedik sor bal szélen kezdődik (nincs behúzás), míg az ötödik megint be van húzva (a harmadik sorral azonos mértékben) :

```
>>> a = 20
>>> if (a > 100):
...     print "a meghaladja a százat"
... else:
...     print "a meghaladja a százat"
...
```

Nyomjunk még egyszer <Enter> -t. A program mégegyszer végrehajtódik és ez alkalommal kiírja:

```
a nem haladja meg a százat
```

Az olvasó már bizonyára kitalálta az **else** (« különben ») utasítás egy alternatív végrehajtás programozását teszi lehetővé, így a programozónak két lehetőség között kell választani. Az **elif** (az « else if » összevonása) utasítást használva még jobb megoldást tudunk adni :

```
>>> a = 0
>>> if a > 0 :
...     print "a pozitív"
... elif a < 0 :
...     print "a negatív"
... else:
...     print "a nulla"
...
```

3.3 Relációs operátorok

Az **if** utasítás után kiértékelt feltétel a következő ***relációs operátorokat*** tartalmazhatja :

```
x == y              # x egyenlő y -nal
x != y              # x nem egyenlő y -nal
x > y               # x nagyobb, mint y
x < y               # x kisebb, mint y
x >= y              # x nagyobb, vagy egyenlő mint y
x <= y              # x kisebb, vagy egyenlő mint y
```

Példa :

```
>>> a = 7
>>> if (a % 2 == 0):
...     print "a páros"
...     print "mert 2-vel való osztása esetén a maradék nulla"
... else:
...     print "a páratlan"
...
```

Jól jegyezzük meg, hogy két érték egyenlőségét a dupla egyenlőségjel operátorral teszteljük, nem pedig az egyszeres egyenlőségjellel[12]. (Az egyszeres egyenlőségjel egy értékadó operátor.) Ugyanezt a szimbolizmust fogjuk találni a *C++* -ban és a *Java*ban).

3.4 Összetett utasítások – Utasításblokkok

Az **if** utasítással használt konstrukció az első ***összetett utasítás***unk. Hamarosan másfélékkel is fogunk találkozni. A Pythonban minden összetett utasításnak mindíg ugyanaz a szerkezete : egy fejsor, ami kettőspontra végződik, ez alatt következik egy vagy több utasítás, ami a fejsor alatt be van húzva. Példa :

```
Fejsor:
    a blokk első utasítása
    ... ...
    ... ...
    a blokk utolsó utasítása
```

Ha a fejsor alatt több behúzott utasítás van, akkor ***mindegyiknek pontosan ugyanannyira kell behúzva lenni*** (például 4 karakterrel kell beljebb lenni). Ezek a behúzott utasítások alkotják az ***utasításblokk***ot. Az utasításblokk : egy logikai együttest képező utasítások sorozata, ami csak a fejsorban megadott feltételek teljesülése esetén hajtódik végre. Az előző bekezdés példájában az **if** utasítást tartalmazó sor alatti két behúzott sor egy logikai blokkot alkot: ez a két sor csak akkor hajtódik végre, ha az **if** -el tesztelt feltétel igaz, vagyis ha a 2-vel való osztás maradéka nulla.

12 Emlékeztető : a % operátor a *modulo* operátor : egy egészosztás maradékát számolja. Így például, **a % 2** a 2-vel való osztás maradékát adja.

3.5 Egymásba ágyazott utasítások

Komplex döntési struktúrák létrehozásához egymásba ágyazható több összetett utasítás. Példa :

```
if torzs == "gerincesek":                          # 1
    if osztaly == "emlősök":                       # 2
        if rend == "ragadozók":                    # 3
            if csalad == "macskafélék":            # 4
                print "ez egy macska lehet"        # 5
        print "ez minden esetre egy emlős"         # 6
    elif osztaly == 'madarak':                     # 7
        print "ez egy kanári is lehet"             # 8
print"az állatok osztályozása összetett"           # 9
```

Elemezzük a példát. A programrészlet csak abban az esetben írja ki az « ez egy macska lehet » -et, ha az első négy feltétel igaz.

Az « ez minden esetre egy emlős » kiírásához szükséges és elégséges, hogy az első két feltétel igaz legyen. Az utóbbi mondatot (6. sor) kiíratót utasítás ugyanazon a behúzási szinten van, mint az if rend == "ragadozók" (3. sor). A két sor tehát ugyanannak a blokknak a része, ami akkor hajtódik végre, ha az 1. és 2. sorban tesztelt feltételek igazak.

Az « ez talán egy kanári » szöveg kiírásához az szükséges, hogy a törzs nevű változó a « gerincesek » értéket, és az osztály nevű változó a « madarak » értéket tartalmazza.

A 9. sor mondatát minden esetben kiírja, mivel ugyanannak az utasításblokknak a része, mint az 1. sor.

3.6 A Python néhány szintaktikai szabálya

Az előzőek alapján összefoglalunk néhány szintaktikai szabályt :

3.6.1 Az utasítások és a blokkok határait a sortörés definiálja

Számos programozási nyelvben minden sort speciális karakterrel kell befejezni (gyakran pontos vesszővel). A Pythonban a sorvégejel[13] játsza ezt a szerepet. (A későbbiekben majd meglátjuk, hogy hogyan hágható át ez a szabály, hogy egy összetett kifejezést több sorra terjesszünk ki. Egy utasítássort kommenttel is befejezhetünk. Egy Python komment mindíg a # karakterrel kezdődik. Ami a # karakter és a LF (linefeed) karakter között van, a fordító figyelmen kívül hagyja.

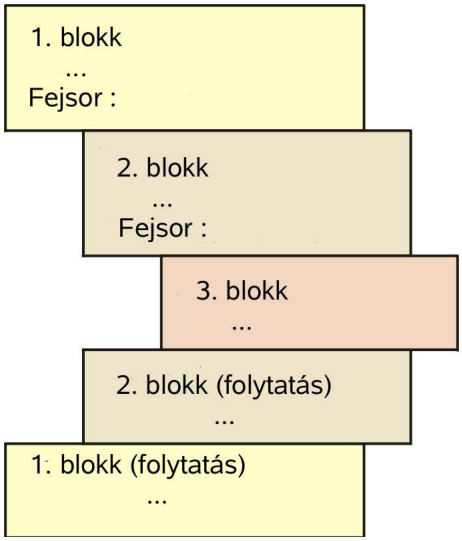
A nyelvek többségében az utasításblokkot speciális jelekkel kell határolni (néha a **begin** és **end** utasításokkal) A *C++ -ban* és *Java -ban*, például, az utasításblokkot kapcsos zárójelekkel kell határolni. Ez lehetővé teszi, hogy az utasításblokkokat egymás után írjunk anélkül, hogy a behúzásokkal és a sorugrásokkal foglalkoznánk. Azonban ez zavaros, nehezen olvasható programok írásához vezethet. Ezért minden programozónak, aki ezeket a nyelveket használja azt tanácsolom, hogy alkalmazzák a sorugrásokat és a behúzásokat **is** a blokkok jó vizuális határolására.

13 Ez a karakter sem a képernyőn, sem a nyomtatott listákon nem jelenik meg. Ennek ellenére jelen van és bizonyos esetekben problémákat okoz, mert nincs azonos módon kódolva a különböző operációs rendszerekben. Erről később fogok beszélni, amikor a szövegfileokról tanulunk (116. oldal)

A Pythonban ***használnunk kell*** a sorugrásokat és a behúzásokat, viszont nem kell más blokkhatároló szimbólumokkal foglalkoznunk. Végeredményben a Python olvasható kódok írására és olyan jó szokások felvételére kényszerít bennünket, amiket meg fogunk őrizni más programozási nyelvek használata során is.

3.6.2 Összetett utasítás = Fej , kettőspont , behúzott utasítások blokkja

A következő fejezettől kezdve sokszor lesz alkalmunk az « utasításblokk » fogalmának az elmélyítésére és gyakorlatok végzésére ebben a tárgykörben.



A szemközti séma összefoglalja az elvet :

- Az utasításblokkok mindíg egy jóldefiniált utasítást tartalmazó fejsorhoz kapcsolódnak (if, elif, else, while, def, ...), ***ami kettőspontra végződik***.
- ***A blokkokat behúzás határolja:*** egyazon blokk minden sorának pontosan ugyanúgy kell behúzva lenni (vagyis ugyanolyan számú betűközzel kell jobbra eltolva lenniük[14]). A behúzásra akármennyi szóközt használhatunk, a programok többsége a 4 többszöröseit alkalmazza.
- Jegyezzük meg, hogy a legkülső blokknak (1. blokk) a bal margón kell lenni (semmibe sincs beágyazva).

3.6.3 A space-eket és a kommenteket rendszerint figyelmen kívül hagyja az interpreter

A sor elején lévő - a behúzásra szolgáló - szóközöktől eltekintve az utasítások és kifejezések belsejében elhelyezett szóközöket majdnem mindíg figyelmen kívül hagyjuk (kivéva, ha ezek egy karakterlánc részét képezik). Ugyanez vonatkozik a kommentekre : ezek mindíg a # karakterrel kezdődnek és az aktuális sor végéig tartanak.

14 Tabulátorokkal is behúzhatunk, de akkor nagyon ügyelni kell arra, hogy azokat követően ugyanabban a blokkban behúzásra ne használjunk hol szóközökeket, hol pedig tabulátorokat. Még ha az eredmény azonosnak is tűnik a képernyőn, a space-ek és a tabulátorok különböző bináris kódúak : a Python ezért ezeket a sorokat úgy fogja tekinteni, hogy különböző képpen vannak behúzva, igy különböző blokkoknak a részei. Ez nehezen debugolható hibákat eredményezhet.
Következésként a programozók többsége a tabulálásokról inkább lemond. Ha egy "intelligens" szövegszerkesztőt használunk, a "Tabuláció helyettesítése szóközzel" opció aktiválásával elkerülhetjük a problémát.

4. Fejezet : Ismétlődő utasítások

4.1 Ismételt értékadás

Még nem jeleztem explicit módon : annyiszor rendelhetünk új értéket egy változóhoz, ahányszor csak akarunk.

Az ismételt értékadás a változó régi értékét egy új értékkel helyettesíti.

```
>>> magassag = 320
>>> print magassag
320
>>> magassag = 375
>>> print magassag
375
```

Ez újra felhívja a figyelmünket arra a tényre, hogy az egyenlőségjelet a Pythonban értékadásra használjuk és hogy semmi esetre sem kell azt összekeverni a matematikában használt egyenlőségjellel. Csábító a **magassag = 320** utasítást egyenlőségre vonatkozó állításként interpretálni, de az nem az !

- Először is , az egyenlőség ***kommutativ,*** míg az értékadás nem az. Így a matematikában az **a = 7** és **7 = a** írása egyenértékű, míg az olyan programutasítás, mint a **375 = magassag** illegális.
- Másodszor, az egyenlőség ***állandó,*** míg az értékadást helyettesíteni lehet, amint azt láttuk. Ha a matematikában egy levezetés elején az **a = b** egyenlőséget állítjuk, akkor az **a** az elkövetkező egész okfejtés alatt állandó marad.
  A programozásban az első értékadás két változó értékeit egyenlővé teheti és egy következő utasítás az után az egyik vagy a másik értékét megváltoztathatja. Példa :

```
>>> a = 5
>>> b = a              # a és b egyenlő értékeket tartalmaz
>>> b = 2              # a és b most különbözőek
```

Megismétlem, a Python lehetővé teszi, hogy több változónak egyidőben adjunk értéket :

```
>>> a, b, c, d = 3, 4, 5, 7
```

A Pythonnak ez a tulajdonsága sokkal érdekesebb, mint amilyennek első pillantásra tűnik. Tegyük fel például, hogy most meg szeretnénk változtatni az **a** és **c** változók értékeit. (Pillanatnyilag **a** értéke 3, és **c** értéke 5. Azt szeretnénk, hogy ez fordítva legyen.) Mit tegyünk ?

(4) Gyakorlat

4.1. Írja meg a szükséges az utasítássorokat, amik a kívánt eredményt adják.

Az gyakorlat során az olvasó biztosan talált egy megoldási módszert, és a tanára valószínűleg meg fogja rá kérni, hogy kommentálja azt az osztályban. Mivel egy megszokott műveletről van szó, ezért a programozási nyelvek gyakran egy rövidítést kínálnak a végrehajtására (például speciális utasításokat, mint a *Basic* nyelv SWAP utasítása). A Pythonban, a *többszörös értékadás* lehetővé teszi ennek a cserének a rendkívül elegáns programozását :

```
>>> a, b = b, a
```

(Természetesen ugyanabban az utasításban más változók értékét is fel tudnánk egyidejűleg cserélni).

4.2 Ciklikus ismétlődések - a while utasítás

Az egyik dolog, amit a számítógépek a legjobban csinálnak, az az azonos feladatok hiba nélküli ismétlése. Az ismétlődő feladatok programozására léteznek eljárások. Az egyik legalapvetőbbel fogjuk kezdeni : a **while** utasítással létrehozott ciklussal.

Írjuk be a következő utasítást :

```
>>> a = 0
>>> while (a < 7):                # (ne felejtsük el a kettőspontot !)
...     a = a + 1                 # (ne felejtsük el a behúzást !)
...     print a
```

Nyomjunk még egyszer <Enter> -t.

Mi történik ?

Mielőtt elolvassa a következő oldal magyarázatát, szánjon rá egy kis időt és írja le a füzetébe ezt az utasítás sorozatot. Írja le a kapott eredményt is és próbálja meg a lehető legrészletesebben megmagyarázni.

Magyarázatok

A **while** jelentése : « amíg ». Ez az utasítás jelzi a Pythonnak, hogy ***az utána következő utasításblokkot mindaddig folyamatosan kell ismételnie,*** amíg az **a** változó tartalma 7-nél kisebb.

Mint az előző fejezetben tárgyalt **if** utasítás, a **while** utasítás is egy ***összetett utasítást*** kezd meg. A kettőspont a sor végén a megismétlendő utasításblokkot vezeti be, aminek kötelezően beljebb igazítva kell lenni. Ahogyan azt az előző fejezetben megtanultuk, egyazon blokk valamennyi utasításának azonos mértékben kell behúzva lenni (vagyis ugyanannyi szóközzel kell jobbra eltolva lenniük).

Megkonstruáltuk tehát első programhurkunkat, ami bizonyos számú alkalommal megismétli a behúzott utasítások blokkját. Ez a következőképpen működik :

- A **while** utasítás esetén a Python a zárójelben levő feltétel kiértékelésével kezdi. (A zárójel opcionális. Csak a magyarázat világossá tétele érdekében használom)
- Ha a feltétel hamis, akkor a következő blokkot figyelmen kívül hagyja és a programvégrehajtás befejeződik[15].
- Ha a feltétel igaz, akkor a Python a ***ciklustestet*** alkotó teljes utasításblokkot végrehajtja, vagyis :
  - az **a = a + 1** utasítást, ami 1-gyel incrementálja az **a** változó tartalmát
    (ami azt jelenti, hogy az **a** változóhoz hozzárendelünk egy új értéket, ami egyenlő az **a** 1-gyel megnövelt előző értékével).
  - a **print** utasítást, ami kiíratja az **a** változó aktuális értékét
- amikor ez a két utasítás végrehajtódott, akkor tanúi voltunk az első **iterráció**nak, és a programhurok, vagyis a végrehajtás visszatér a **while** utasítást tartalmazó sorra. Az ott található feltételt újra kiértékeli és így tovább.
  Ha példánkban, az **a < 7** *feltétel* még igaz, a ciklustest újra végrehajtódik és folytatódik a ciklus.

Megjegyzések :

- A feltételben kiértékelt változónak a kiértékelést megelőzően léteznie kell. (Egy értéknek kell már hozzárendelve lenni.)
- Ha a feltétel eredetileg hamis, akkor a ciklustest soha sem fog végrehajtódni.
- Ha a feltétel mindíg igaz marad, akkor a ciklustest végrehajtása vég nélkül ismétlődik (de legalábbis addig, amíg a Python maga működik). Ügyelni kell rá, hogy a ciklustest legalább egy olyan utasítást tartalmazzon, ami a **while**-lal kiértékelt feltételben megváltoztatja egy beavatkozó változó értékét úgy, hogy ez a feltétel hamissá tudjon válni és a ciklus befejeződjön.

Végtelen ciklus példája (kerülendő) :

```
>>> n = 3
>>> while n < 5:
...     print "hello !"
```

15 ... legalább is ebben a példában. Később meg fogjuk látni, hogy a programvégrehajtás a behúzott blokkot követö első utasítással folytatódik és hogy ez ugyanannak az utasításblokknak képezi a részét, mint amelyiknek a while utasítás része.

4.3 Táblázatkészítés

Kezdjük újra az első gyakorlattal, de az alábbi kis módosítással:

```
>>> a = 0
>>> while a < 12:
...     a = a +1
...     print a , a**2 , a**3
```

Az 1-től 12-ig terjedő számok négyzetének és köbének listáját kell megkapnunk.
Jegyezzük meg, hogy a **print** utasítás lehetővé teszi, hogy ugyanabba a sorba több kifejezést írassunk ki, egyiket a másik után : elég vesszővel elválasztani őket. A Python automatikusan beszúr egy szóközt a kiírt elemek közé.

4.4 Egy matematikai sor megalkotása

Az alábbi kis program a « Fibonacci sor » első hat elemét íratja ki. Egy olyan számsorról van szó, amelynek minden tagja egyenlő az őt megelőző két tag összegével. Elemezzük ezt a (***többszörös értékadás***t helyesen alkalmazó) programot. Írjuk le a lehető legjobban az utasítások szerepét.

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...     print b,
...     a, b, c = b, a+b, c+1
```

Amikor elindítjuk a programot, a következőt kapjuk :

```
1 2 3 5 8 13 21 34 55 89
```

A *Fibonacci* sor tagjai ugyanabba a sorba vannak kiírva. Ezt annak a vesszőnek köszönhetjük, amit a **print** utasítást tartalmazó sor végére írtunk. Ha töröljük a vesszőt, akkor a számok egymás alá lesznek írva.

Az olvasó nagyon gyakran fog a programjaiban olyan ciklusokat írni, mint amilyeneket itt elemzünk. Fontos dologról van szó, amit tökéletesen kell tudni kezelni. Legyen benne biztos, a gyakorlatok segítségével fokozatosan el fog jutni odáig.

Amikor egy ilyen természetű problémát vizsgálunk, természetesen meg kell néznünk az utasításssorokat, de főként a ciklusban érintett ***különböző változók egymást követő állapotait*** kell elemeznünk. Ez korántsem mindíg egyszerű. Hogy segítsek ezt világosabbá tenni, vegyük a fáradtságot és rajzoljunk egy, az alábbihoz hasonló, állapottáblát a « Fibonacci sor » programunknak :

Változók	a	b	c
Kezdő értékek	1	1	1
Az iterráció során egymás után felvett értékek	1 2 3 5 ...	2 3 5 8 ...	2 3 4 5 ...
Helyettesítő kifejezések	b	a+b	c+1

Egy ilyen táblázatban kézzel végezzük el a számítógép munkáját, sorról-sorra megadva azokat az értékeket, amiket az egyes változók az egymást követő iterrációkban föl fognak venni. Azzal kezdjük, hogy a táblázat tetejére felírjuk az érintett változók nevét. A következő sorba ezen változók kezdő értékei kerülnek (azok az értékek, amikkel a ciklus indulása előtt rendelkeztek). Végül legalulra azok a kifejezések kerülnek, amiket minden egyes ciklusban az egyes változók állapotának módosítására használtunk.

Kitöltjük az első iterrációknak megfelelő néhány sort. Egy sor értékeinek a meghatározásához elég az előző sor értékeit, és az egyes oszlopok alján található helyettesítő kifejezést felhasználni. Így ellenőrizzük, hogy amit kapunk az a keresett eredmény e. Ha nem ez a helyzet, akkor más helyettesítő kifejezéseket kell kipróbálni.

Gyakorlatok :

4.2. Írjon egy programot, ami kiíratja a 7-es szorzótábla első 20 tagját.

4.3. Írjon egy programot, ami euróban kifejezett pézösszegeket kanadai dollárba vált át és az eredményt egy táblázatba írja ki. A táblázatban a pézösszegek « geometriai haladvány » szerint növekedjenek úgy, mint az alábbi példában :

```
1 euro = 1.65 dollar
2 euro = 3.30 dollar
4 euro = 6.60 dollar
8 euro = 13.20 dollar
```

stb. (16384 euronál kell megállni)

4.4. Írjon egy programot, ami kiír egy 12 számból álló sorozatot, aminek minden tagja vagy egyenlő az előző taggal, vagy annak háromszorosa.

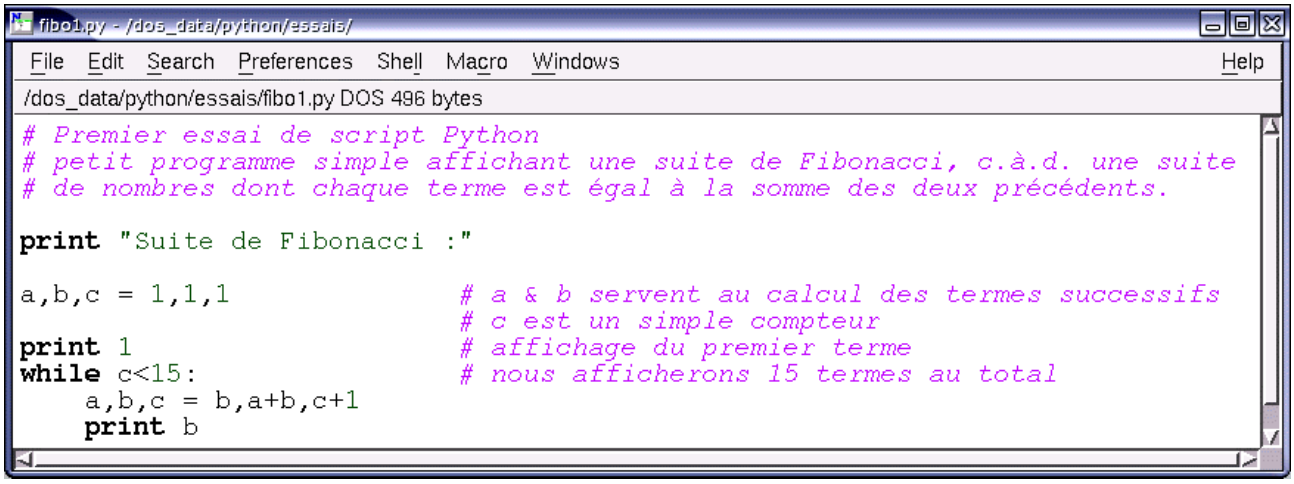
4.5 Az első scriptek, avagy : Hogyan őrizzük meg programjainkat ?

Eddig a Pythont mindíg *interaktív módban* használtuk (vagyis az utasításokat minden alkalommal közvetlenül az interpreterbe írtuk be anélkül, hogy azokat később egy fileba mentettük volna). Ez lehetőséget adott arra, hogy közvetlen kísérletezéssel nagyon gyorsan megtanuljuk a nyelv alapjait. Ez az eljárás egy nagy kényelmetlenséggel jár : minden utasítássorozat, amit beírtunk helyrehozhatatlanul eltűnik amikor lezárjuk az interpretert. Mielőtt tovább haladnánk megtanuljuk programjaink merev lemezen vagy floppyn lévő fileba való mentését, hogy egymást követő fázisokban át tudjuk őket dolgozni, más gépekre át tudjuk őket vinni, stb.

Ehhez mostantól fogva valamilyen szövegszerkesztővel fogjuk az utasítás sorozatainkat szerkeszteni (például Linux alatt *Joe*-val, *Nedit*-tel, *Kate*-tel, ... *MS-DOS* alatt *Edit*-tel, ... *Windows* alatt *Wordpad* -del, vagy még jobb egy olyan fejlesztő környezetbeli szövegszerkesztő, mint az *IDLE* vagy a *PythonWin*). Tehát írunk egy ***script***-et, amit aztán elmenthetünk, módosíthatunk, másolhatunk, stb., mint bármilyen más szöveget, amit a számítógéppel kezelünk[16].

16 Szövegszerkesztőt azzal a feltétellel használhatnánk, hogy a mentést « tiszta szövegként » (***plain text***) végeznénk (laptördelési tag-ek nélkül). Kívánatosabb azonban egy valódi « inteligens » ANSI szövegszerkesztőt használni, mint amilyen a *nedit* vagy az *IDLE*. Ezek a Python szintaxisának megfelelően színezik a forrásszöveget, ami segíti a szintaxishibák elkerülését. Az *IDLE*-ben : File → New window (vagy CTRL-N) -val megnyitunk egy új ablakot, amibe a scriptünket fogjuk írni. A végrehajtáshoz a script mentése után : Edit → Run script (vagy CTRL-F5).

Az alábbi ábra a Nedit használatát illusztrálja Gnome (Linux) alatt :



Utána, amikor szeretnénk a programunkat tesztelni, elég lesz elindítani a Python interpretert, argumentumként a scriptet tartalmazó file nevét megadva. Például, ha az « EnScriptem » nevű file-ba mentettünk el egy scriptet, akkor elég lesz a következő parancsot beírni egy terminálablakba, hogy a script végrehajtódjon :

```
python EnScriptem
```

Hogy még jobban csináljuk, adjunk a filenak olyan nevet, ami **.py** kiterjesztésre végződik.

Ha tiszteletben tartjuk ezt a konvenciót, akkor (*Windows, KDE, Gnome,* alatt) a filemanagerben (Explorer a *Windows* vagy *Konqueror* a *KDE* alatt) a filenévre vagy a megfelelő ikonra kattintva elindíthatjuk a script végrehajtását. Ezek a grafikus filemanagerek tudják, hogy minden alkalommal, amikor a felhasználó megpróbál megnyitni egy **.py** kiterjesztésű filet, el kell indítsák a Python interpretert. (Ez természetesen azt feltételezi, hogy ezek megfelelően voltak konfigurálva.) Ugyanez a konvenció ezen kívül lehetővé teszi az « intelligens » szövegszerkesztőknek a Python scriptek automatikus felismerését és a szintaktikus színezés alkalmazását.

Egy Python script olyan utasítás szekvenciákat fog tartalmazni, mint amilyenekkel eddig kísérleteztünk Mivel ezeket az utasítássorokat arra szánjuk, hogy megőrizzük és később akár magunk, akár mások újraolvassák, ezért ***melegen ajánlott, hogy scriptjeinket a lehető legjobban szövegezzük meg, bőségesen kommentezzük***. A programozás fő nehézsége a korrekt algoritmusok elkészítése. Ahhoz, hogy az algoritmusokat jó feltételek mellett tudjuk ellenőrizni, javítani, módosítani, stb., lényeges, hogy szerzőjük a lehető legteljesebben és legvilágosabban írja le őket. Ezeknek a leírásoknak a legjobb helye maga a script (így azok nem veszhetnek el).

Egy jó programozó mindíg ügyel arra, hogy sok kommentet szúrjon be a scriptjeibe. Ezzel az eljárással nemcsak az esetleges más olvasóknak könnyíti meg az algoritmusai olvasását, de magát is kényszeríti arra, hogy még világosabb gondolatai legyenek.

Egy scriptbe szinte bárhova bármilyen kommentet beszúrhatunk. Elég eléjük tenni egy # karaktert. Amikor a Python interpreter rátalál erre a karakterre, ami utána következik azt figyelmen kivülhagyja az aktuális sor végéig.

Értsük meg, hogy fontos, hogy a programozás ***menete során*** írjunk kommenteket. Ne várjuk meg a script befejezését, hogy majd utána írjuk hozzá azokat. Az olvasó lassanként rá fog jönni, hogy a programozó rendkívül sok időt tölt a saját kódja újraolvasásával (azért, hogy módosítsa, hibát

keressen benne, stb.) Ezt az újraolvasást nagyban egyszerűsíti a számos magyarázat és megjegyzés.

Nyissuk meg a szövegszerkesztőnket és írjuk be az alábbi scriptet :

```
# Első Python script
# Fibonacci-sort írat ki, azaz egy olyan számsort, aminek minden tagja
# az előző két tag összege.

a, b, c = 1, 1, 1           # a & b az egymást követő tagok számolására valók
                            # c egy számláló
print 1                     # az első tag kiíratása
while c<15:                 # összesen 15 tagot íratunk ki
    a, b, c = b, a+b, c+1
    print b
```

Azért, hogy rögtön jó példát mutassak, a scriptet a program működésének rövid leírását tartalmazó három kommentsorral kezdjük. Váljon ez az olvasónak is szokásává a saját scriptjeinél.

A kódsorok dokumentálva vannak. Ha úgy jár el, ahogyan én, vagyis a megfelelő utasítások jobboldalára szúr be kommenteket, akkor ügyeljen rá, hogy eléggé el legyenek tolva az utasításoktól azért, hogy ne zavarják azok olvashatóságát.

Amikor alaposan ellenőriztük a szövegünket mentsük el és hajtassuk végre.

Megjegyzés : Bár nem feltétlenül szükséges, de mégegyszer azt javaslom, hogy a scriptjeinknek **.py** kiterjesztésre végződő fileneveket adjunk. Ez sok segítséget jelent egy folderben történő azonosításuknál. A grafikus filekezelők (*Windows Explorer*, *Konqueror)* egyébként ezt a kiterjesztést használják arra, hogy egy speciális ikont kapcsoljanak hozzájuk. Viszont kerüljük az olyan nevek választását, amik már létező python moduloknak a nevei. Olyan neveket például, mint *math.py* vagy *Tkinter.py* tilos használni !

Ha *Linux* alatt szövegmódban dolgozunk, vagy *MSDOS* ablakban, scriptünket a **python** *scriptnév* utasítással hajtathatjuk végre. Ha *Linux* alatt grafikus módban dolgozunk, megnyithatunk egy terminálablakot és ugyanígy járhatunk el. A *Windows Explorer*-ben vagy a *Konqueror*-ban scriptünk végrehajtását a megfelelő ikonra történő egérkattintással indíthatjuk.

Ha az *IDLE*-lel dolgozunk, a szerkesztés alatt a <Ctrl-F5> billentyű kombináció segítségével indíthatjuk scriptünket. Konzultáljon tanárával az esetleges más indítási lehetőségekről más operációs rendszerekben.

4.6 Ékezetes és speciális karakterekre vonatkozó megjegyzés :

A 2.3 verziótól kezdve a francia nyelvet használóknak ajánlatos minden Python scriptjük elejére beírni a következő pszeudo-commentek egyikét (kötelezően az első vagy a második sorba) :

```
# -*- coding:Latin-1 -*-
```

vagy :

```
# -*- coding:Utf-8 -*-
```

Ezek a pszeudo-commentek azt jelzik a Pythonnak, hogy a scriptben :

- vagy a fő nyugat-európai nyelvek ékezetes karakterkészletét használjuk (francia, olasz, portugál, stb.) egy byteon kódolva az ISO-8859 norma szerint;

- vagy a Unicodenak nevezett két byteos kódolást használjuk (aminek az Utf-8 változata csak a « speciális » karaktereket kódolja két byteon, a standard ASCII karakterek egy byte-on vannak kódolva). Ez az utóbbi rendszer egyre jobban kezd elterjedni, mivel lehetővé teszi mindenféle eredetű (görög, arab, ciril, japán, stb.) karakter együttlétezését ugyanabban a dokumentumban.

A Python mindkét rendszert tudja használni, de meg kell neki adnunk, hogy melyiket használjuk. Ha az operációs rendszerünk úgy van konfigurálva, hogy a billentyűleütések Utf-8 kódokat generálnak, akkor konfiguráljuk úgy a szövegszerkesztőnket, hogy az is ezt a kódot használja és tegyük a fönt megadott második pszeudo-commentet minden scriptünk elejére.

Ha az operációs rendszerünk a régi norma (ISO-8859) szerint működik, akkor inkább az elsö pszeudo-commentet kell használnunk.

Ha semmit sem adunk meg, akkor időnként figyelmeztető üzeneteket fogunk kapni az interpretertől és esetleg némi nehézségeket fogunk tapasztalni amikor az IDE környezetben szerkesztjük scriptjeinket (speciálisan Windows alatt).

- Függetlenül attól, hogy egyik, vagy másik normát, vagy egyiket sem használjuk, a scriptünk korrekt módon fog végrehajtódni. Ahhoz, hogy a saját rendszerünkön tudjuk a kódot szerkeszteni, a megfelelő opciót kell választani.

Gyakorlatok :

4.5. Írjon egy programot, ami kiszámolja egy derékszögű parallelepipedon térfogatát, aminek meg van adva a szélessége, a magassága és a hosszúsága.

4.6. Írjon egy programot, ami átszámolja a kiindulásként megadott egészszámú másodpercet évekké, hónapokká, napokká, percekké és másodpercekké.
(Használja a modulo operátort : %).

4.7. Írjon egy programot, ami kiíratja a 7-es szorzótábla első 20 tagját, csillaggal jelölve azokat, amelyek 3-nak többszörösei.
Példa : 7 14 21 * 28 35 42 * 49

4.8. Írjon egy programot, ami kiszámolja 13-as szorzótábla első 50 tagját, de csak azokat írja ki, melyek 7-nek többszörösei.

4.9. Írjon egy programot, ami a következő jelsorozatot írja ki :

```
*
**
***
****
*****
******
*******
```

5. Fejezet : A fő adattípusok

A 2. fejezetben már kezeltünk különböző típusú adatokat : egész és valós számokat és karakterláncokat. Ideje mostmár egy kicsit közelebbről is megvizsgálni ezeket az adattípusokat és másokat is felfedezni.

5.1 Numerikus adatok

Az eddigi gyakorlatokban már használtunk két adattípust : ***egész*** és ***valós*** számokat (utóbbiakat lebegőpontos számoknak is nevezik). Próbáljuk meg bemutatni ezek jellemzőit (és korlátait) :

5.1.1 Az« integer » és« long » típusok

Tegyük fel, hogy úgy szeretnénk módosítani az előző *Fibonacci* sorozatos gyakorlatunkat, hogy több tagot írassunk ki. *A priori* elegendő a második sorban megváltoztatni a ciklus feltételét. A « **`while c<49:`** » feltétellel 49 tagot kell kapnunk. Módosítsuk tehát a gyakorlatot úgy, hogy kiíratjuk a fővátozó típusát is :

```
>>> a, b, c = 1, 1, 1
>>> while c<49:
        print c, " : ", b, type(b)
        a, b, c = b, a+b, c+1
...
...
...  (az első 43 tag kiíratása)
...
44  :  1134903170 <type 'int'>
45  :  1836311903 <type 'int'>
46  :  2971215073 <type 'long'>
47  :  4807526976 <type 'long'>
48  :  7778742049 <type 'long'>
```

Mi állapítható meg ?

Ha nem használnánk a **type()** függvényt, ami minden egyes iterráció alkalmával lehetővé teszi a **b** változó típusának ellenőrizését, akkor semmit sem vennénk észre. A Fibonacci számok sorozatát minden probléma nélkül kiíratja a script (és még számos taggal meghosszabbíthatnánk a sort).

Úgy tűnik tehát, hogy a Python bármilyen méretű egész számot képes kezelni.

Ennek ellenére az előző gyakorlat jelzi, hogy « valami » történt amikor ezek a számok nagyon naggyá váltak. A program elején az **a**, **b** és **c** változók implicit módon ***integer*** típusúaknak vannak definiálva. Mindíg ez történik a Pythonban, amikor egy egész számot rendelünk egy változóhoz, feltéve, hogy a szám nem túl nagy. A számítógép memóriájában ez az adattípus egy 4 byte-ból (vagy 32 bitből) álló blokk formájában van kódolva. Márpedig a 4 byte-on kódolt decimális értékek tartománya -2147483648 -től + 2147483647 -ig terjed. (Lásd az általános informatikai kurzust).

Az ilyen típusú számokkal végzett számolások mindíg nagyon gyorsak, mert a számítógép processzora az ilyen 32 bites egész számokat közvetlenül képes kezelni. Viszont, ha nagyobb számokat, vagy valós számokat (« lebegőpontos » számok) kell kezelni, akkor a programoknak, (az interpretereknek és a compilereknek) jelentős kódolási/dekódolási munkát kell végezniük, hogy végül a processzornak csak maximum 32 bites bináris egészeken történő műveleteket küldjenek.

Azt már tudjuk, hogy a Python dinamikusan definiálja a változóinak a típusát.

Mivel a leghatékonyabb típusról van szó (mind számolási sebességben, mind memóriafoglalásban kifejezve), ezért a Python alapértelmezésben minden alkalommal, amikor csak lehet az ***integer*** típust használja, vagyis ha a kezelt értékek a fentebb már említett határok között vannak (kb. ± 2 milliárd).

Ha a kezelt értékek ezeken a határokon kívül esnek, akkor a kódolásuk bonyolultabbá válik a számítógép memóriájában. Azok a változók, amelyekhez ilyen számokat rendelünk, automatikusan « hosszú egészeknek » lesznek definiálva (ezt a típust a Python terminológiában ***long***-nak nevezzük)

Ez a ***long*** típus az egész értékek majdnem végtelen pontosságú kódolását teszi lehetővé. Egy ilyen formában definiált érték akárhány szignifikáns számjeggyel rendelkezhet, mivel ezt a számot ***csak a számítógép memóriájának mérete limitálja***.

Példa :

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
        print c, ": ", b
        a, b, c = b, a*b, c+1

1 :  2
2 :  6
3 :  12
4 :  72
5 :  864
6 :  62208
7 :  53747712
8 :  3343537668096
9 :  179707499645975396352
10 :  6008587943056673222701555425185792
11 :  1079788315649669138143849229447384578592430704390307 84
12 :  648800305446607527907368373691049776950010342842280428918276494561862 34
582611607420928
13 :  700566989011183200292376413995762169216245450579726979173836923132717 54
883621235064434673400268965204696103008832506249008437424702378475 52
14 :  454528076456265799856362940482493512051682398707229461514016556556583 98
642227616335815123825782460196980206141536747116094173550514227947953005917 00
969504226930790382476340558291752968319462245039335017547760330040127583682 56
>>>
```

A fenti példában a kiírt számok mérete nagyon gyorsan nő, mert mindegyikük az előző két szám szorzatával egyenlő.

Kiinduláskor az **a**, **b** és **c** ***integer*** típusúak, mert kis egész értékeket : 3, 2 és 1: rendelünk hozzájuk. A 8. iterrációtól kezdve azonban a **b** és **a** automatikusan ***long*** típusúakká lesznek alakítva : a 6-os és 7-es tagok szorzásának eredménye már jóval a fentebb említett 2 milliárdos határ fölött van.

A sor egyre nagyobb számokkal folytatódik, de a számolás sebessége csökken. A ***long*** típusúként tárolt számok méretüktől függően változó mennyiségű helyet foglalnak el a számítógép memóriájában.

5.1.2 A« float »típus

A « valós szám », vagy « lebegő pontos » numerikus adattípussal, amit angolul « *floating point number* »-nek neveznek és amit emiatt ***float*** -nak fogunk nevezni a Pythonban, már találkoztunk.

Ez a típus nagyon nagy vagy nagyon kis számokkal (például tudományos adatokkal) történő számolásokat tesz lehetővé állandó pontossággal.

Ahhoz, hogy a Python egy numerikus adatot ***float*** típusúnak tekintsen, elegendő, ha a szám egy tizedes pontot, vagy 10 -nek egy hatványkitevőjét tartalmazza.

Például a következő értékeket :

```
3.14      10.      .001      1e100      3.14e-10
```

a Python automatikusan ***float*** típusúként értelmezi.

Próbáljuk ki ezt az adattípust egy új programocskában (amit az előző inspirált) :

```
>>> a, b, c = 1., 2., 1             # => a és b 'float' típusúak lesznek
>>> while c <18:
...     a, b, c = b, b*a, c+1
...     print b

2.0
4.0
8.0
32.0
256.0
8192.0
2097152.0
17179869184.0
3.6028797019e+16
6.18970019643e+26
2.23007451985e+43
1.38034926936e+70
3.07828173409e+113
4.24910394253e+183
1.30799390526e+297
         Inf
         Inf
```

Ez alkalommal is egy olyan sorozatot íratunk ki, melynek tagjai nagyon gyorsan növekednek, lévén minden tag az előző két tag szorzata. A nyolcadik taggal messze meghaladjuk egy ***integer*** kapacitását. A kilencedik taggal a Python automatikusan áttér a tudományos jelölésmódra (« e+n » jelentése : -szer tíz az n-ediken). A tizenötödik tag után újra egy (hibaüzenet nélküli) átmenet tanui vagyunk, a valóban nagyon nagy számokat egyszerűen « inf » (« végtelen ») jelöli.

A ***float*** típus a 10^{-308} és 10^{308} közé eső (pozitív vagy negatív) számok 12 értékes számjegy pontossággal történő manipulációját teszi lehetővé. Ezek a számok speciálisan 8 byteon (64 biten) vannak kódolva a gép memóriájában : a kód egy része a 12 értékes számjegynek felel meg, a másik része a nagyságrendnek (10 hatványának).

(5) Gyakorlatok :

5.1. Írjon egy programot, ami a kiindulásul fokokban, percekben és másodpercekben megadott szögeket radiánba számolja át.

5.2. Írjon egy programot, ami a kiindulásul radiánokban megadott szögeket fokokba, percekbe és másodpercekbe számolja át.

5.3. Írjon egy programot, ami Celsius fokokba számolja át a kiindulásul Fahrenheit fokokban kifejezett hőmérsékletet és a fordított irányú átalakítást is elvégzi. .
Az átalakítás képlete : $T_F = T_C \times 1{,}8 + 32$.

5.4. Írjon egy programot, ami a bankban elhelyezett 4,3 % -os kamatozású tőke 20 év alatt felhalmozódott évi kamatait számolja ki.

5.5. Egy régi indiai legenda szerint a sakkjátékot egy öreg bölcs találta ki. A király meg akarta azt neki köszönni és azt mondta, hogy jutalmul bármilyen ajándékot megad érte. Az öreg azt kérte, hogy adjon neki egy kevés rizset öreg napjaira, pontosan annyi szem rizset, hogy az általa feltalált játék első kockájára 1 szemet, 2 második kockára kettőt, a harmadikra négyet, és így tovább egészen a 64-ik kockáig.
Írjon egy Python programot, ami kiíratja a sakktábla 64 kockájának mindegyikére elhelyezett rizsszemek számát. Számolja ki ezt a számot kétféleképpen
- a rizsszemek pontos száma (egész szám)
- a rizsszemek száma tudományos jelölésmódban (valós szám)

5.2 Az alfanumerikus adatok

Eddig csak számokat kezeltünk. Egy számítógépprogram ugyanígy kezelhet betűkaraktereket, szavakat, mondatokat, vagy bármilyen szimbólumsorozatot. A programozási nyelvek többségében erre a célra létezik egy ***karakterlánc*** (vagy angolul ***string***) nevű adatszerkezet.

5.2.1 A« string » (karakterlánc) típus

A Pythonban ***string*** típusú adat bármilyen karaktersorozat, amit vagy ***szimpla idézőjelek*** (aposztrof), vagy ***dupla idézőjelek*** határolnak

Példák :

```
>>> mondat1 = 'a kemény tojást.'
>>> mondat2 = '"Igen", válaszolta,'
>>> mondat3 = "nagyon szeretem"
>>> print mondat2, mondat3, mondat1
"Igen", válaszolta, nagyon szeretem a kemény tojást.
```

A 3 változó : mondat1, mondat2, mondat3 tehát ***string*** típusú változó.

Jegyezzük meg, hogy az olyan stringeket, melyekben aposztrofok vannak, idézőjelekkel határoljuk, míg az idézőjeleket tartalmazó stringeket aposztrofokkal határoljuk. Azt is jegyezzük meg, hogy a **print** utasítás a kiírt elemek közé egy betűközt szúr be.

A « \ » (*backslash*) néhány kiegészítő finomságot tesz lehetővé :

- Lehetővé teszi, hogy egy parancsot, ami nem fér el egy sorban, azt több sorba írjunk (ez bármilyen típusú parancsra érvényes).
- Egy karakterlánc belsejében a backslash speciális karakterek (sorugrás, aposztrofok, dupla idézőjelek, stb.) beszúrását teszi lehetővé. Példák :

```
>>> txt3 = "E\'meg kicsoda ? kérdezte."
>>> print txt3
"E'meg kicsoda ?" kérdezte.
>>> hello = "Ez egy hosszú sor\n ami több szövegsort\
... tartalmaz (Azonos módon \n működik, mint a C/C++.\n\
...     Jegyezzük meg,hogy a white space-ek\n a sor elején lényegesek.\n"
>>> print hello
Ez egy hosszú sor
 ami több szövegsort tartalmaz (Azonos módon
 működik, mint a C/C++.
    Jegyezzük meg,hogy a white space-ek
 a sor elején lényegesek..
```

Megjegyzések :

- A **\n** egy sorugrást idéz elő.
- A **\'** lehetővé teszi, hogy aposztrofokkal határolt karakterláncba aposztrofot szúrjunk be.
- Mégegyszer megismétlem : a kis/nagybetű lényeges a változónevekben (Szigorúan tiszteletben kell tartanunk a kezdeti kis/nagybetű választást).

« Háromszoros idézőjelek » :

Hogy egy karakterláncba könnyebben szúrjunk be speciális vagy « egzotikus » karaktereket anélkül, hogy a backslasht alkalmaznánk, vagy magát a backslasht tudjuk beszúrni, a karakterláncot ***háromszoros aposztroffal*** vagy ***háromszoros idézőjellel*** határolhatjuk :

```
>>> a1 = """
... Használat: izee[OPTIONS]
... { -h
...   -H host
... }"""

>>> print a1

Használat: izee[OPTIONS]
{ -h
  -H host
}
```

5.2.2 Hozzáférés egy karakterlánc egyes karaktereihez

A karakterláncok az ***összetett adatok***nak nevezett általánosabb adattípus egy speciális esetét képezik. Egy összetett adat egy olyan entitás, ami egyszerűbb entitások együttesét egyetlen struktúrában egyesíti : egy karakterlánc esetében például ezek az egyszerűbb entitások nyilván maguk a karakterek. A körülményektől függően a karakterláncot hol mint egyetlen objektumot, hol mint különálló karakterek együttesét akarjuk kezelni. Egy olyan programozási nyelvet, mint a Pythont tehát el kell látni olyan eljárásokkal, amik lehetővé teszik egy karakterlánc egyes karaktereihez való hozzáférést. Látni fogjuk, ez nem olyan bonyolult :

Egy karakterláncot a Python a ***szekvenciák*** kategória egy objektumának tekint. A szekvenciák elemek rendezett együttesei. Ez egyszerűen azt jelenti, hogy egy string karakterei mindíg egy bizonyos sorrendben vannak elrendezve. Következésként a string minden egyes karakterének meghatározható a szekvenciabeli helye egy index segítségével.

Ahhoz, hogy egy adott karakterhez hozzáférjünk, a karakterláncot tartalmazó változó neve után szögletes zárójelbe írjuk a karakter stringbeli pozíciójának megfelelő numerikus indexet.

Figyelem : amint azt egyebütt ellenőrizhetjük, az informatikában az adatokat majdnem mindíg ***nullától kezdve*** számozzuk (nem pedig egytől). Ez a helyzet egy string karakterei esetében :

Példa :

```
>>> ch = "Stéphanie"
>>> print ch[0], ch[3]
S p
```

5.2.3 Elemi műveletek karakterláncokon

A Pythonnak számos, karakterláncok kezelésére szolgáló függvénye van (kis/nagybetűs átalakítás, rövidebb karakterláncokra való darabolás, szavak keresése, stb.). A későbbiekben el fogunk mélyedni ebben a tárgykörben (lásd 129. oldalt).

Pillanatnyilag megelégedhetünk azzal, hogy tudjuk, külön-külön hozzáférhetünk egy karakterlánc minden egyes eleméhez, amint ezt fentebb magyaráztam. Tudjunk róla, hogy a következőket is megtehetjük :

- több rövid karakterláncból összerakhatunk egy hosszabbat. Ezt a műveletet összekapcsolásnak (***concatenatio-***nak) nevezzük és a + operátort használjuk rá a Pythonban. (Ez az operátor számokra alkalmazva az összeadás műveletére, míg karakterláncokra alkalmazva az összekapcsolásra szolgál).
- Példa :

```
a = 'A kis halból'
b = ' nagy hal lesz'
c = a + b
print c
A kis halból nagy hal lesz
```

- meghatározhatjuk egy karakterlánc hosszát (vagyis a karakterek számát) a **len()** függvény hívásával :

```
>>> print len(c)
29
```

- Egy számot reprezentáló karakterláncot számmá alakíthatunk.

  Példa :

```
>>> ch = '8647'
>>> print ch + 45
   ==> *** error *** nem adhatunk össze egy stringet és egy számot
>>> n = int(ch)
>>> print n + 65
8712                          # OK : 2 számot összeadhatunk
```

  Ebben a példában az **int()** belső függvény a stringet számmá alakítja. A **float()** függvény segítségével valós számmá lehet alakítani egy karakterláncot.

Gyakorlatok :

5.6. Írjon egy programot, ami meghatározza, hogy egy karakterlánc tartalmazza-e az « e » karaktert.

5.7. Írjon egy programot, ami megszámolja az « e » karakter előfordulásainak számát egy stringben.

5.8. Írjon egy programot, ami egy új változóba másol át egy karakterláncot úgy, hogy csillagot szúr be a karakterek közé.

5.9. Így például, « `gaston` »-ból « `g*a*s*t*o*n` » lesz.

5.10. Írjon egy programot, ami egy új változóba fordított sorrendben másolja át egy karakterlánc karaktereit.
Így például « `zorglub` » -ból « `bulgroz` » lesz.

5.11. Az előző gyakorlatból kiindulva írjon egy scriptet, ami meghatározza, hogy egy karakterlánc palindrom e (vagyis ami mindkét irányból olvasva ugyan az), mint például « radar » vagy « sós ».

5.3 A listák (első megközelítés)

Az előző fejezetben tárgyalt stringek az összetett adatokra voltak az első példák. Az ***összetett adatokat*** arra használjuk, hogy struktúráltan csoportosítunk értékegyütteseket. Lépésről-lépésre fogjuk megtanulni más összetett adattípusok használatát : a ***listák***-ét, a ***tuple***-két és a ***szótárak***-ét (***dictionnair***)[17]. E három adattípus közül most csak az elsőt fogjuk elég röviden tárgyalni. Egy meglehetősen terjedelmes témáról van szó, amire többször vissza kell majd térnünk.

A lista definíciója a Pythonban : ***szögletes zárójelbe zárt, vesszővel elválasztott elemek csoportja***. Példa :

```
>>> nap = ['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
>>> print nap
['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
```

Ebben a példában **nap** nevű változó értéke egy lista.

Megállapíthatjuk : a választott példában a listát alkotó elemek különböző típusúak lehetnek. Az első három elem string, a negyedik egész, az ötödik valós típusú, stb. (A későbbiekben látni fogjuk, hogy egy lista maga is lehet egy listának eleme !). Ebben a tekintetben a lista fogalma meglehetősen különbözik a tömb (***array***) vagy az « indexelt változó » fogalmától, amivel más programozási nyelvekben találkozunk.

Jegyezzük meg, hogy a listák is szekvenciák, úgy mint a karakterláncok, vagyis objektumok rendezett csoportjai. A listát alkotó különböző elemek mindíg ugyanabban a sorrendben vannak elrendezve, mindegyikükhöz külön hozzá tudunk férni, ha ismerjük a listabeli indexüket. Ezeknek az indexeknek a számozása nullától indul, nem pedig egytől, a karakterláncokhoz hasonlóan.

Példák :

```
>>> nap = ['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
>>> print nap[2]
szerda
>>> print nap[4]
20.357
```

A stringektől (amik egy nem módosítható adattípust képeznek /erre a későbbiekben többször lesz alkalmunk visszatérni/) eltérően egy listának meg lehet változtatni az elemeit :

```
>>> print nap
['hétfő', 'kedd', 'szerda', 1800, 20.357, 'csütörtök', 'péntek']
>>> nap[3] = nap[3] +47
>>> print nap
['hétfő', 'kedd', 'szerda', 1847, 20.357, 'csütörtök', 'péntek']
```

17 Majd létrehozhatjuk a saját adattípusainkat is, amikor megtanuljuk az osztály (***class***) fogalmát (lásd 161. oldal)

A lista meghatározott elemét a következő módon helyettesíthetjük más elemmel :

```
>>> nap[3] = 'Július'
>>> print nap
['hétfő', 'kedd', 'szerda', 'Július', 20.357, 'csütörtök', 'péntek']
```

A **len()** belső függvény, amivel már a stringeknél találkoztunk, a listákra is alkalmazható. A listában levő elemek számát adja vissza :

```
>>> len(nap)
7
```

Egy másik belső függvény - a **del()** [18] - segítségével (az indexe alapján) bármelyik elemet törölhetjük a listából :

```
>>> del(nap[4])
>>> print  nap
['hétfő', 'kedd', 'szerda', 'Július', 'csütörtök', 'péntek']
```

Ugyanígy lehetőség van arra, hogy hozzáfűzzünk egy elemet egy listához, de ahhoz, hogy ezt megtegyük meg kell gondolni, hogy a lista egy objektum, aminek az egyik metódusát fogjuk használni. Az *objektum* és *metódus* fogalmakat a későbbiekben fogom megmagyarázni, de most meg tudom mutatni, hogy ez hogyan működik egy lista speciális esetében :

```
>>> nap.append('szombat')
>>> print nap
['hétfő', 'kedd', 'szerda', 'Július', 'csütörtök', 'péntek', 'szombat']
>>>
```

A fenti példa első sorában az **append()** *metódust* alkalmaztuk a **'szombat'** *argumentummal* a **nap** *objektumra.* Ha emlékeztetek arra, hogy az *append* szó hozzáfűzést jelent, akkor érthető, hogy az **append()** *metódus* egy olyan függvényfajta, ami valamilyen módon a « lista » típusú objektumokhoz van kapcsolva, vagy az objektumokba van integrálva. A függvénnyel használt argumentum természetesen az az elem, amit a lista végéhez akarunk fűzni.

A későbbiekben meg fogjuk látni, hogy egész sor ilyen *metódus* van (vagyis olyan függvények amik « lista » típusú objektumokba vannak integrálva, vagy inkább « becsomagolva »). Jegyezzük meg, hogy ***egy metódust úgy alkalmazunk egy objektumra, hogy egy ponttal kapcsoljuk őket össze***. (Elől áll annak a változónak a neve, ami egy objektumra hivatkozik, utána a pont, majd a metódus neve, ez utóbbit mindíg egy zárójelpár követi).

18 Léteznek olyan technikák listák feldarabolására, elemcsoportok beszúrására, elemek eltávolítására, stb., amik egy sajátos szintaxist használnak, amelyben csak az indexek fordulnak elő. Ezeket a (karakterláncokra is alkalmazható) technikákat általánosan *slicing*-nek *(*szeletelésnek) hívjuk. Úgy használjuk őket, hogy a szögletes zárójelbe több indexet írunk. Így a nap[1:3] a ['kedd', 'szerda'] alcsoportot jelöli. Ezeket a speciális technikákat egy kicsit később fogom leírni (lásd a 132. és az azt követő oldalakat).

A karakterláncokhoz hasonlóan a listákkal is részletesen fogok foglalkozni a későbbiekben (lásd a 133. oldalt). Ennek ellenére elég ismeretünk van ahhoz, hogy el tudjuk kezdeni őket használni a programjainkban. Elemezzük például az alábbi kis scriptet és magyarázzuk meg a működését :

```
nap = ['hétfő', 'kedd', 'szerda', 'csütörtök', 'péntek', 'szombat']
a, b = 0, 0
while a<25:
     a = a + 1
     b = a % 7
     print a, nap[b]
```

Az 5. sorban a « *modulo* » operátort használjuk, amivel már előzőleg találkoztunk és ami jó szolgálatokat tehet a programozásban. Számos nyelvben (a Pythonban is) a % jel reprezentálja. Milyen műveletet hajt végre ez az operátor ?

Gyakorlatok :

5.12. Legyenek adottak a következő listák :

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Január', 'Február', 'Március', 'Április', 'Május', 'Június',
 'Július', 'Augusztus', 'Szeptember', 'Október', 'November', 'December']
```

Írjon egy kis programot, ami egy új **t3** listát hoz létre. Ennek felváltva kell tartalmazni a két lista minden elemét úgy, hogy minden hónap nevét követnie kell a megfelelő napok számának : `['Január',31,'Február',28,'Március',31, stb...]`.

5.13. Írjon egy programot, ami kiíratja egy lista összes elemét. Ha például a fenti gyakorlat t2 listájára alkalmaznánk, akkor a következőt kellene kapnunk :

```
Január Február Március Április Május Június Július Augusztus Szeptember
Október November December
```

5.14. Írjon egy programot, ami megkeresi egy adott lista legnagyobb elemét. Például, ha a `[32, 5, 12, 8, 3, 75, 2, 15]`, listára alkalmaznánk, akkor a következőt kellene kiírnia :

```
a lista legnagyobb elemének az értéke 75.
```

5.15. Írjon egy programot, ami megvizsgálja egy számlista minden elemét (például az előző példa listáját) azért, hogy két új listát hozzon létre. Az egyik csak az eredeti lista páros számait tartalmazza, a másik a páratlanokat. Például, ha a kiindulási lista az előző gyakorlat listája, akkor a programnak egy **páros listát** kell létrehoznia, ami a `[32, 12, 8, 2]` -t tartalmazza és egy **páratlan listát** ami `[5, 3, 75, 15]` -t tartalmazza. Trükk : Gondoljon az előzőekben említett modulo (%) operátor használatára !

5.16. Írjon egy programot, ami egy szavakból álló lista elemeit egyenként megvizsgálja azért, hogy két új listát hozzon létre. (például: ['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra'] Az egyikben 6 karakternél rövidebb szavakat legyenek, a másikban 6, vagy annál több karaktert tartalmazzonó szavak legyenek.

6. Fejezet : Előre definiált függvények

A programozásban az egyik legfontosabb fogalom a ***függvény***[19]fogalma. A függvények lehetővé teszik, hogy egy összetett programot egyszerűbb alprogramokra bontsunk szét, amiket aztán megint kisebb darabokra szedhetünk szét és így tovább. Másrészt a függvények újra felhasználhatók : ha van egy függvényünk, ami például négyzetgyököt tud számolni, azt a programjainkban mindenütt újra felhasználhatjuk anélkül, hogy újra kellene írnunk minden egyes alkalommal.

6.1 Interakció a felhasználóval : az input() függvény

A kidolgozott scriptek többsége igényli egyik vagy másik pillanatban a felhasználó beavatkozását (egy paraméter beírását, egy kattintást az egyik gombra, stb.). Egy szövegmódú scriptben (mint amilyeneket eddig írtunk) a legegyszerűbb módszer az **input()** belső függvény alkalmazása. Ez a függvény a futó program megszakítását eredményezi. A felhasználót felhívja, hogy írjon be a billentyűzettel karaktereket és <Enter> -rel fejezze be. Amikor lenyomja az <Enter> -t a program végrehajtása folytatódik és a függvény a felhasználó által beírt értéket adja meg visszatérési értékül. Ezt az értéket bármilyen változóhoz hozzá lehet rendelni.

Hívhatjuk az **input()** függvényt úgy, hogy a zárójelet üresen hagyjuk. Elhelyezhetünk benne a felhasználó számára egy magyarázó üzenetet is. Például :

```
print 'Írjon be valamilyen egész számot : ',
nn = input()
print nn, 'négyzete', nn**2
```

vagy még :

```
keresztnev = input('Írja be a keresztnevét (idézőjelben) : '
print 'Jó napot,', keresztnev
```

Fontos megjegyzések :

- Az **input()** függvény egy olyan értéket ad vissza, aminek a típusa a felhasználó által beírt érték típusának felel meg. Példánkban az **nn** változó egy egészet, egy stringet, egy valós számot, stb. fog tartalmazni aszerint, hogy hogyan fog a felhasználó dönteni. Ha a felhasználó egy stringet akar beírni, akkor azt ***aposztrofok vagy idézőjelek között*** kell beírnia. A későbbiekben majd meglátjuk, hogy egy jó scriptnek mindíg ellenőrizni kell, hogy a beírt érték típusa megfelel-e annak, amit a programban várunk.
- Ezért scriptjeinkben gyakran a hasonló funkciójú **raw_input()** függvényt fogjuk előnyben részesíteni, ami mindíg egy ***karakterláncot*** ad vissza. Ezt a karakterláncot az után az **int()** és **float ()** függvények segítségével számmá alakíthatjuk. Példa :

```
>>> a = raw_input('Írjon be egy adatot : ')
Írjon be egy adatot : 52.37
>>> type(a)
<type 'str'>
>>> b = float(a)        # átalakítás numerikus értékké
>>> type(b)
<type 'float'>
```

19 A Pythonban a "függvény" szót különbségtétel nélkül használjuk egyszer a valódi függvények, máskor az **eljárások** (**procedure**) jelölésére A későbbiekben meg fogjuk határozni a különbséget a két közeli fogalom között.

6.2 Függvénymodul importálása

Már találkoztunk a nyelvbe ***beépített függvények*** fogalmával, mint amilyen például a **len()** függvény, ami a karakterlánc hosszát adja meg. Magától értetődő, hogy nem lehet az összes elképzelhető függvényt a Python standardba belevenni, mivel ezekből végtelen sok van : egyébként hamarosan meg fogjuk tanulni, hogyan hozzuk létre magunk új függvényeket. A nyelvbe beépített függvények száma viszonylag csekély : ezek azok, amik nagyon gyakran használhatók. A többiek ***moduloknak*** nevezett külön file-okban vannak csoportosítva.

A modulok tehát file-ok, amik függvénycsoportokat fognak egybe. A későbbiekben majd meglátjuk, hogy kényelmesebb egy nagyméretű programot több kisebb méretű részre felbontani, hogy egyszerűsítsük a karbantartást. Egy jellegzetes Python-alkalmazás tehát egy főprogramból és egy vagy több modulból áll, melyek mindegyike kiegészítő függvények definícióit tartalmazza.

Nagyszámú modult adnak hivatalosan a Pythonnal. Más modulokat más szolgáltatóknál találhatunk. Gyakran egymással rokonságban lévő függvénycsoportokat próbálnak meg ugyanabba a modulba összefogni, amit ***könyvtárnak*** nevezünk.

A **math** modul például számos olyan matematikai függvény definícióját tartalmazza, mint a *sinus, cosinus, tangens, négyzetgyök*, stb. Ezeknek a függvényeknek a használatához elég a scriptünk elejére beszúrni a következő sort :

```
from math import *
```

Ez a sor jelzi a Pythonnak, hogy az aktuális programba bele kell venni a *math* modul minden függvényét (ezt jelzi a *). Ez a modul egy matematikai függvénykönyvtárat tartalmaz.

A scriptbe például az alábbiakat írjuk:

gyok **= sqrt(szam)** ez a *gyok* nevű változóhoz rendeli a *szam* négyzetgyökét,
sinusx = sin(szog) ez a *sinusx* nevű változóhoz rendeli a *szog* (radiánban !) sinus -át , stb.

Példa :

```
# Demo : a <math> modul függvényeinek használata

from math import *

szam = 121
szog = pi/6   # azaz 30° (a math könyvtár tartalmazza a pi definícióját is)
print 'négyzetgyök', szam, '=', sqrt(szam)
print 'sinus ', szog, 'radian', '=', sin(szog)
```

A script végrehajtásakor a következőket írja ki :

```
négyzetgyök 121 = 11.0
sinus 0.523598775598 radian = 0.5
```

Ez a rövid példa már nagyon jól illusztrálja a függvények fontos jellemzőit :

- a függvény ***valamilyen név és a hozzá kapcsolt zárójelek*** formájában jelenik meg
  példa : **`sqrt()`**
- a zárójelekben egy vagy több ***argumentumot adunk át*** a függvénynek
  példa : **`sqrt(121)`**
- a függvénynek van egy ***visszatérési értéke*** (azt is mondjuk, hogy « visszaad » egy értéket)
  példa : **`11.0`**

Mindezeket a következő oldalakon fogom kifejteni. Ezek a matematikai függvények csak az első példák a függvényekre. Egy pillantást vetve a Python könyvtárak dokumentációjába megállapíthatjuk, hogy mostantól fogva már számos függvény áll rendelkezésünkre, nagyszámú feladat -beleértve nagyon összetett matamatikai algoritmusokat - megoldására (a Pythont általánosan használják az egyetemeken magas szintű tudományos problémák megoldására). Nem fogom e függvények részletes listáját megadni. Egy ilyen lista könnyen hozzáférhető a Python helprendszerében :

HTML dokumentáció→ Python dokumentáció → Modul index → math

A következő fejezetben megtanuljuk, hogy hogyan hozhatunk létre magunk függvényeket.

(6) Gyakorlatok :

*(Megjegyzés : Mindegyik gyakorlatban használja adatbevitelre a **raw_input()** függvényt !)*

6.1. Írjon egy programot, ami m/sec és km/h -ba számolja át a felhasználó által mérföld/h -ban megadott sebességet. . (*1 mérföld = 1609 méter*)

6.2. Írjon egy programot, ami kiszámolja a kerületét és a területét annak a háromszögnek, melynek 3 oldalát a felhasználó adja meg.
(*Ismétlés : egy háromszög területét a következő formula segítségével számoljuk ki :*
$S=\sqrt{d\cdot(d-a)\cdot(d-b)\cdot(d-c)}$
ahol d a kerület felét, a, b, c az oldalak hosszát jelöli).

6.3. Írjon egy programot, ami kiszámolja egy adott hosszúságú matematikai inga periódusidejét
A periódusidő számolására szolgáló formula a következő: $T=2\,\pi\sqrt{\frac{l}{g}}$,
ahol : *l* az inga hossza és *g* a szabadesés gyorsulása a kísérlet helyén.

6.4. Írjon egy programot, ami értékeket tesz egy listába. Ennek a programnak ciklusban kell működni úgy, hogy mindaddig kéri az értékeket, amíg a felhasználótól úgy nem dönt, hogy befejezésként <Entert> üt. A program a lista kiírásával fejeződik be. Működési példa :

```
Írjon be egy értéket : 25
Írjon be egy értéket : 18
Írjon be egy értéket : 6284
Írjon be egy értéket :
[25, 18, 6284]
```

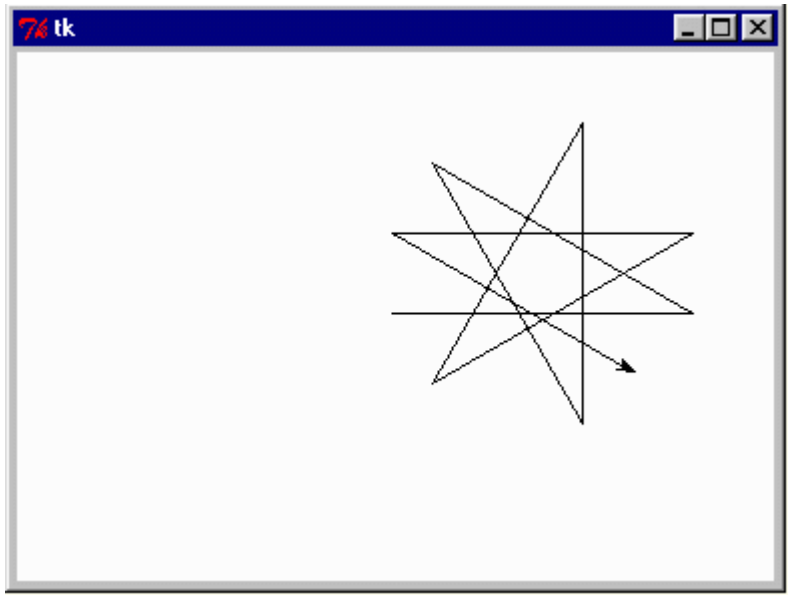
6.3 Egy kis pihenő a turtle (teknős) modullal

Láttuk, a Python egyik nagyszerű tulajdonsága, hogy különböző ***modulok*** importálásával rendkívül egyszerűen lehet hozzáadni számos új funkcionalitást.

Ennek illusztrációjaként most egy kicsit más objektumokkal fogunk szórakozni, mint a számok. Meg fogunk vizsgálni egy Python modult, ami « teknős grafikák » létrehozását teszi lehetővé, vagyis olyan geometriai rajzokét, mint amilyeneket egy kis virtuális « teknős » hagy maga után, aminek az elmozdulásait a monitoron egyszerű utasításokkal vezérelhetjük.

A teknős aktiválása gyerekjáték. A hosszú magyarázat helyett inkább próbáljuk ki máris :

```
>>> from turtle import *
>>> forward(120)
>>> left(90)
>>> color('red')
>>> forward(80)
```



A gyakorlat nyilván sokkal beszédesebb, ha ciklusokat használunk :

```
>>> reset()
>>> a = 0
>>> while a <12:
        a = a +1
        forward(150)
        left(150)
```

Figyelem : mielőtt elindítunk egy ilyen scriptet, mindíg ellenőrizzük, hogy nem tartalmaz-e végtelen ciklust (lásd a 37. oldalt), mert ellenkező esetben azt kockáztatjuk, hogy nem tudjuk visszavenni a vezérlést (speciálisan Windows alatt).

Írjunk olyan scripteket, amik előre megadott ábrákat rajzolnak. A *turtle* modulban a következő főbb függvények állnak rendelkezésünkre :

reset()	Mindent töröl és újrakezd
goto(x, y)	Az x, y koordinátájú helyre megy
forward(tavolsag)	Egy adott távolságot előre megy
backward(tavolsag)	Egy adott távolságot visszafele megy
up()	Felemeli a ceruzát (hogy rajzolás nélkül tovább tudjon menni)
down()	Leteszi a ceruzát (hogy újrakezdje a rajzolást)
color(szin)	<szin> egy előre definiált string lehet ('red', 'blue', 'green', stb.)
left(szog)	Adott (fokokban kifejezett) szöggel balra fordul
right(szog)	Jobbra fordul
width(vastagsag)	Vonalvastagságot választ
fill(1)	Zárt körvonal kitöltése egy kiválasztott színnel
write(texte)	a <texte> -nek "-lel vagy '-fal határolt karakterláncnak kell lenni

6.4 Egy kifejezés igaz/hamis értéke

Ha egy program olyan utasításokat tartalmaz, mint a **while** vagy az **if,** akkor a számítógépnek ki kell számolni egy feltétel igazságértékét, vagyis meg kell határoznia, hogy a kifejezés igaz e vagy hamis. Például a **while c<20:** -szal bevezetett ciklus mindaddig végrehajtódik, amíg a **c<20** feltétel ***igaz*** marad.

De hogyan tudja egy számítógép meghatározni, hogy valami igaz, vagy hamis ?

Azt már tudjuk, hogy a számítógép csak számokat kezel. Mindent, amit a számítógépnek kezelni kell először mindíg át kell alakítani numerikus értékké. Ez érvényes az igaz/hamis fogalomra is. A Pythonban éppen úgy, mint a ***C***-ben, a ***Basic***-ben és számos más programnyelvben minden nullától különböző numerikus értéket « igaz »-nak tekintünk. Csak a nulla érték a « hamis ». Példa :

```
a = input('Írjon be egy tetszőleges számot')
if a:
    print "igaz"
else:
    print "hamis"
```

A fenti kis script csak akkor ír ki « hamis »-at, ha 0 értéket írunk be. Minden más numerikus értékre « igaz »-at kapunk.

Ha egy karakterláncot, vagy egy listát írunk be, akkor is « igaz »-at kapunk. Csak az üres stringeket vagy az üres listákat tekinti « hamis »-nak.

Az előzőek azt jelentik, hogy egy olyan kiértékelendő kifejezést, mint az **a > 5** feltétel, a számítógép először numerikus értékké alakít át. (Általában 1-gyé, ha a kifejezés igaz és 0-vá, ha a kifejezés hamis) :

```
a = input(' Írjon be egy numerikus értéket : ')
b = (a < 5)
print 'A b értéke', b, ':'
if b:
    print "A b feltétel igaz"
else:
    print " A b feltétel hamis"
```

A fenti script a **b = 1** értéket küldi vissza (a feltétel igaz), ha 5-nél kisebb értéket írtunk be.

Ezek csak előzetes információk a Boole algebrának nevezett logikai műveletekkel kapcsolatban. A későbbiekben meg fogjuk tanulni, hogy a bináris számokra olyan operátorokat alkalmazhatunk, mint az **and**, **or**, **not** stb., ami lehetővé teszi, hogy ezekkel a számokkal összetett logikai műveleteket végezzünk.

6.5 Ismétlés

A következőkben nem fogunk új fogalmakat tanulni, egyszerűen az eddig tanultakat fogjuk kis programok készítésére alkalmazni.

6.5.1 Az utasításfolyam vezérlése – Egyszerű lista használata

Kezdjük a feltételes elágaztatásokkal (bármely nyelvnek talán a legfontosabb utasításcsoportjáról van szó !) :

```
# Lista és feltételes elágaztatás használata

print "A script három szám közül a legnagyobbat keresi"
print 'Írjon be három, vesszővel elválasztott számot : '
# Megjegyzés : a list() függvény az argumentumként átadott adatszekvenciát
# listává alakítja. Az alábbi utasítás tehát a felhasználó által átadott
# adatokat az nn listává alakítja:
nn = list(input())
max, index = nn[0], 'első'
if nn[1] > max:              # ne felejtsük el a kettőspontot !
    max = nn[1]
    index = 'második'
if nn[2] > max:
    max = nn[2]
    index = 'harmadik'
print "Ezen számok közül a legnagyobb", max
print "Ez a szám a(z) ", index, ". a listában"
```

Megjegyzés : Ebben a gyakorlatban megint találkozunk a 3. és 4. fejezetben már bőségesen kommentált « utasításblokk » fogalmával, amit feltétlenül meg kell tanulnunk. Ismétlésül : az utasításblokkokat ***behúzások*** határolják. Az első **if** utasítás után például két behúzott sor van, amik egy utasításblokkot definiálnak. Ezek az utasítások csak akkor hajtódnak végra, ha az **nn[1] > max** feltétel igaz.

A következő sor (az amelyik a második **if** utasítást tartalmazza) viszont nincs behúzva. Ez a sor tehát ugyanazon a szinten van, mint azok a sorok, melyek a program gerincét definiálják. Ezeknek a soroknak az utasítás tartalma mindíg végrehajtódik, míg a következő két sor (ami egy másik blokkot alkot) csak akkor hajtódik végre, ha az **nn[2] > max** feltétel igaz.

Azonos logikát követve látjuk, hogy a két utolsó sor a főblokk része és így mindíg végrehajtódik.

6.5.2 A while ciklus- Beágyazott utasítások

Folytassuk más struktúrák beágyazásával :

```
# Összetett utasítások <while> - <if> - <elif> - <else>                  #  1

print 'Válasszon egy számot 1-től 3-ig (vagy nullát befejezésként) '     #  3
a = input()                                                             #  4
while a != 0:           # a != operátor jelentése "nem egyenlő "        #  5
    if a == 1:                                                          #  6
        print "Ön az egyet választotta "                                #  7
        print "első, egyedi, egység ..."                                #  8
    elif a == 2:                                                        #  9
        print "Ön a kettőt szereti   :"                                 # 10
        print "pár, páros, duo ..."                                     # 11
    elif a == 3:                                                        # 12
        print "Ön a három közül a legnagyobb mellett dönt:"             # 13
        print "trio, hármas, triplet ..."                               # 14
    else :                                                              # 15
        print "1 és 3 közötti számot legyen szíves"                     # 16
    print 'Válasszon egy számot 1-től 3-ig (vagy 0 befejezésként)     ' # 17
    a = input()                                                         # 18
print "Ön nullát írt be :"                                              # 19
print "A gyakorlatnak vége van."                                        # 20
```

Itt egy **while** ciklussal találkozunk, amibe egy **if, elif** és **else** utasításcsoport van beágyazva. Most is figyeljük meg, hogy a program logikai struktúráját behúzások segítségével alakítottuk ki (és ne felejtsük el a « : » -ot egyik fejsor végéről se !)

A **while** utasítást itt arra használjuk, hogy a felhasználó válasza után újra kezdjük a kérdésfeltevést (hacsak a felhasználó úgy nem dönt, hogy kilép egy nulla beírásával. (Emlékeztetőül : a **!=** operátor jelentése « nem egyenlő ») A ciklusmagban találjuk az **if, elif** és **else** utasításcsoportot (a 6.-16. sorokban), ami a programvégrehajtást különböző válaszok felé irányítja, majd egy print és egy **input()** utasítás következik (17. és 18. sor) amiket mindíg végrehajt a program. Vegyük észre, hogy ugyanaddig vannak behúzva, mint az **if, elif** és **else** blokkja. Ezek után az utasítások után a programhurok és a végrehajtás visszatér a **while** utasításra (5. sor). A két utolsó **print** utasítás (19. és 20. sor) csak a ciklusból való kilépés után hajtódik végre.

Gyakorlatok

6.5. Mit csinál az alábbi program abban a négy esetben, melyben előre meghatároztuk, hogy az a változó értéke: 1, 2, 3 vagy 15 ?

```
if a !=2:
 print 'vesztett'
elif a ==3:
 print 'egy kis türelmet kérek'
else :
 print 'nyert'
```

6.6. Mit csinálnak ezek a programok ?

a)
```
a = 5
 b = 2
 if (a==5) & (b<2):
 print '"&" jelentése "és"; az "and" szót is használhatjuk '
```
b)
```
a, b = 2, 4
 if (a==4) or (b!=4):
 print 'nyert'
 elif (a==4) or (b==4):
 print 'majdnem nyert'
```
c)
```
a = 1
 if not a:
 print ' nyert'
 elif a:
 print 'vesztett
```

Hajtassuk végre a c) programot a = 0 -val a = 1 helyett. Mi történik ? Következtessen !

6.7. Vegyük a c) programot a = 0 -val a = 1 helyett. Mi történik ? Következtessen !

6.8. Írjon egy programot, ami adott a és b egész korlátok esetén összeadja a 3 **és** 5 korlátok közé eső többszöröseit.
Vegyük például az a=0, b=32 -t → az eredménynek 0 +15 +30 = 45 -nek kell lenni.
Módosítsa úgy a programot, hogy az adja össze a 3-nak **vagy** az 5-nek az a és b határok közé eső többszöröseit. A 0 és 32 határokkal az eredménynek : 0 + 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 + 20 + 21 + 24 + 25 + 27 + 30 = 225 -nek kell lenni.

6.9. Határozzuk meg, hogy egy év szökőév vagy sem. (Az A év szökőév ha A osztható 4-gyel. Viszont nem az, ha A többszöröse 100-nak, kivéve, ha A 400 -nak többszöröse).

6.10. Kérje a felhasználótól a nevét és a nemét (F vagy N) . Ezektől az adatoktól függően írassa ki a felhasználó nevét és « Úr » -at vagy « Asszony » -t.

6.11. Kérjük meg a felhasználót, hogy írjon be három hosszúságadatot: a, b, c -t. Ennek a három hosszúságnak a segítségével határozza meg, hogy lehet-e egy háromszöget szerkeszteni. Majd határozza meg, hogy ez a háromszög: derékszögű, egyenlőszárú, egyenlőoldalú vagy általános háromszög. Figyelem : egy derékszögű háromszög lehet egyenlőszárú.

6.12. Kérjük meg a felhasználót, hogy írjon be egy egész számot. Ez után írassa ki ennek a számnak vagy a négyzetgyökét, vagy egy üzenetet, ami jelzi, hogy ennek a számnak a négyzetgyökét nem lehet kiszámolni.

6.13. Convertáljuk az iskolai N pontszámot, amit a felhasználó ad meg (például 85-ből 27) egy standardizált jeggyé a következő feltételek szerint :

Pont	Értékelés
N >= 80 %	A
80 % > N >= 60 %	B
60 % > N >= 50 %	C
50 % > N >= 40 %	D
N < 40 %	E

6.14. Legyen a következő felsorolás egy lista :
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', Maximilien', 'Alexandre-Benoît', 'Louise']
Írjon egy scriptet, ami kiírja ezen nevek mindegyikét és a karaktereik számát.

6.15. Írjon egy programhurkot, ami a felhasználótól kéri a tanulók érdemjegyeit. A hurok csak akkor fejeződjön be, ha a felhasználó egy negatív értéket ír be. Az így beírt jegyekkel hozzon létre egy listát. Minden új jegy beírása után (tehát minden iterrációnál) írassa ki a beírt jegyek számát, a legnagyobb és a legkisebb jegyet és a jegyek átlagát.

6.16. Írjon egy scriptet, ami két 10000 kg tömegű test között ható gravitációs erő értékét íratja ki. A testek közötti távolságok 5 cm -től (0,05 m) kezdődő geometriai sorozatot alkotnak, melynek kvóciense 2.

A gravitációs erőt a következő összefüggés alapján számoljuk : $F = 6{,}67\ 10^{-11} \cdot \frac{m \cdot m'}{d^2}$

Példa :

```
d = .05 m :  ez erő nagysága  2.668 N
d = .1 m  :  ez erő nagysága  0.667 N
d = .2 m  :  ez erő nagysága  0.167 N
d = .4 m  :  ez erő nagysága  0.0417 N
stb.
```

7. Fejezet : Saját függvények

A programozás annak a művészete, hogy a számítógépet olyan feladatok elvégzésére tanítjuk meg, amiket előzőleg nem tudott végrehajtani. Az egyik legérdekesebb erre szolgáló módszer az, hogy felhasználói függvények formájában új utasításokat illesszünk az általunk használt programozási nyelvbe.

7.1 Függvény definiálása

Az eddig írt scriptjeink mind nagyon rövidek voltak, mivel a célunk csak az volt, hogy elsajátítsuk a nyelv elemeit. Amikor majd valódi projekteket kezdünk fejleszteni, gyakran rendkívül bonyolult problémékkal fogunk találkozni és a programsorok száma sokasodni fog.

Egy probléma hatékony megközelítése gyakran a problémának több, egyszerűbb alproblémára való felbontásából (dekompozíciójából) áll, amiket azután külön vizsgálunk. (Ezek az alproblémák esetleg tovább bonthatók még egyszerűbb alproblémákra és így tovább). Fontos, hogy az algoritmusokban[20] pontosan mutassuk be ezt a dekompozíciót, hogy azok világosak maradjanak.

Másrészt gyakran előfordul, hogy ugyanazt az utasítás sorozatot többször kell alkalmaznunk egy programban és nyilván nem kívánjuk azt rendszeresen megismételni.

A ***függvények***[21] és az ***objektumosztályok*** eltérő alprogram-struktúrák, amiket a magasszintű nyelvek alkotói azért találtak ki, hogy a fent említett nehézségeket megoldják. A Python függvénydefiníciójának leírásával kezdjük. Az objektumokat és osztályokat (class-okat) később fogjuk tanulmányozni.

Már találkoztunk különböző előre programozott függvényekkel. Most lássuk, hogyan definiáluk mi magunk új függvényeket.

A függvénydefiníció szintaxisa a Pythonban következő :

```
def függvényNeve(paraméterlista):
    ...
    utasításblokk
    ...
```

- Függvénynévnek a nyelv foglalt[22] szavai kivételével bármilyen nevet választhatunk azzal a feltétellel, hogy semmilyen speciális vagy ékezetes karaktert sem használhatunk (az aláhúzás karakter « _ » megengedett). A változónevekhez hasonlóan főként a kisbetűk használata javasolt, nevezetesen a szavak elején (a nagybetűvel kezdődő szavakat fenn fogjuk tartani az *osztályok* számára, amiket a későbbiekben fogunk tanulmányozni).
- A **def** utasítás az **if** -hez és a **while** -hoz hasonlóan egy *összetett utasítás*. Az a sor, amelyik ezt az utasítást tartalmazza kötelezően kettősponttal végződik, ami egy utasításblokkot vezet be, amit nem szabad elfelejtenünk behúzni.
- A *paraméterlista* határozza meg, hogy argumentumként milyen információkat kell megadni, ha

20 Algoritmusnak egy probléma megoldására szolgáló műveletek részletes felsorolását nevezzük.

21 Más nyelvekben léteznek **rutinok** (ezeket néha alprogramoknak hívják) és **procedurák** (eljárások)
A Pythonban nincsenek **rutinok**. A **függvény** elnevezés egyszerre jelöli a szigorú értelemben vett függvény fogalmát (aminek visszatérési értéke van) és a proceduráét (aminek nincs visszatérési értéke).

22 A Python foglalt szavainak teljes listája a 23. oldalon található.

majd használni akarjuk a függvényt. (A zárójel üresen is maradhat, ha a függvénynek nincs szüksége argumentumokra).

- Egy függvényt gyakorlatilag úgy használunk, mint akármilyen más utasítást. A programtörzsben a ***függvényhívás*** a függvény nevéből és az azt követő zárójelekből áll.
  Ha szükséges, a zárójelben adjuk meg azokat az argumentumokat, amiket át akarunk adni a függvénynek. Elvileg a függvénydefinícióban megadott mindegyik paraméter számára meg kell adni egy argumentumot, bár adhatók alapértelmezett értékek ezeknek a paramétereknek (lásd később).

7.1.1 Paraméterek nélküli egyszerű függvény

A függvények első konkrét tárgyalásakor megint interaktív módban fogunk dolgozni. A Python interaktív üzemmódja ideális olyan kisebb tesztek végzésére, mint amilyenek most következnek. Ez egy olyan könnyebbség, amit nem minden programozási nyelv kínál fel !

```
>>> def tabla7():
...     n = 1
...     while n <11 :
...         print n * 7,
...         n = n+1
...
```

Ennek a néhány sornak a beírásával egy nagyon egyszerű függvényt definiáltunk, ami kiszámolja és kiíratja a 7-es szorzótábla első tíz tagját. Figyeljük meg a zárójeleket[23], a kettős pontot, és a fejsort követő utasításblokk (ez az az utasításblokk, ami a függvénytörzset alkotja) behúzását.

Ahhoz, hogy a függvényt használjuk, elég ha a nevével hívjuk. Így:

```
>>> tabla7()
```

kiíratja :

- 7 14 21 28 35 42 49 56 63 70

23 Egy függvénynevet mindíg zárójelnek kell követni, még akkor is, ha a függvény semmilyen paramétert sem használ. Ez egy írásbeli konvencióból ered, ami kiköti, hogy bármely számítógép programozással foglalkozó szövegben egy függvénynevet mindíg egy üres zárójelpárnak kell követni.
Ezt a konvenciót követjük ebben a könyvben.

Ezt a függvényt annyiszor használhatjuk, ahányszor akarjuk. Belevehetjük egy másik függvény definíciójába, mint az alábbi példában :

```
>>> def tabla7tripla():
...     print 'A 7-es szorzótábla három példányban :'
...     tabla7()
...     tabla7()
...     tabla7()
...
```

Hívjuk ezt a függvényt a :

```
>>> tabla7tripla()
```

utasítás begépelésével. A következő kiírást kell eredményként kapnunk :

```
A 7-es szorzótábla három példányban :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Az első függvény tehát hívhat egy másodikat, ami egy harmadikat hívhat és így tovább. Talán még nem látja az olvasó ennek a hasznát, de két érdekes tulajdonságot már megfigyelhet :

- Egy új függvény létrehozása lehetőséget ad egy utasításcsoport elnevezésére. Így egy másodlagos, összetett algoritmus egyetlen parancs mögé rejtésével - aminek nagyon explicit nevet adhatunk - egyszerűsíthetjük a program gerincét.
- Egy új függvény létrehozása az ismétlődő kódrészek kiszöbölése révén a program egyszerűsítését szolgálhatja. Például, ha ugyanabban a programban többször ki kell íratni a 7-es szorzótáblát, akkor nem kell minden alkalommal újraírni az algoritmust, ami elvégzi ezt a munkát.

Egy függvény tehát egy új testreszabott utasítás, amit mi magunk szabadon adunk hozzá a programozási nyelvünkhöz.

7.1.2 Paraméteres függvény

Az utolsó példáinkban egy függvényt definiáltunk és alkalmaztunk a 7-es szorzótábla tagjainak kiíratására. Tegyük fel, hogy ugyanezt szeretnénk megtenni a 9-es szorzótáblával. Természetesen írhatunk egy új függvényt erre, de ha később a 13-as szorzótáblára van szükségünk, akkor megint újrakezdhetjük. Nem volna érdemesebb egyetlen olyan függvényt definiálni, ami kérésre akármelyik szorzótáblát ki tudná íratni ?

Amikor hívjuk a függvényt, természetesen meg kell tudnunk neki adni, hogy melyik szorzótáblát kívánjuk kiíratni. Azt az információt, amit a függvénynek a hívásakor adunk át: ***argumentumnak*** nevezzük. Már többször találkoztunk olyan beépített függvényekkel, amik argumentumot használnak. A **sin(a)** függvény például az **a** szög sinus-át számolja. A **sin(a)** függvény tehát az **a** numerikus értéket használja argumentumként.

Egy ilyen függvény definíciójában előre gondoskodnunk kell egy speciális változóról, ami az átadott argumentumot fogadja. Ezt a speciális változót ***paraméternek*** nevezzük. Ennek a változónak a szokásos szintaktikai szabályoknak megfelelően választunk nevet (nem tartalmazhat ékezetes karaktereket, stb.) és a függvénydefiníciót követő zárójelbe írjuk.

A minket érdeklő esetben a következőt kapjuk :

```
>>> def tabla(alap):
...     n = 1
...     while n <11 :
...        print n * alap,
...        n = n + 1
```

A **tabla()** függvény, ahogy azt fent definiáltuk az **alap** paramétert használja a megfelelő szorzótábla első tíz tagjának a kiszámolására.

Az új függvény teszteléséhez elég, ha azt egy argumentummal hívjuk. Példák :

```
>>> tabla(13)
13 26 39 52 65 78 91 104 117 130

>>> tabla(9)
9 18 27 36 45 54 63 72 81 90
```

Ezekben a példákban a függvény hívásakor zárójelben megadott érték (az argumentum) automatikusan az **alap** nevű paraméterhez van rendelve. A függvénytörzsben az **alap** ugyanazt a szerepet játsza, mint bármely más változó. Amikor beírjuk a **tabla(9)** utasítást, jelezzük a gépnek, hogy a **tabla()** függvényt akarjuk végrehajtatni az **alap** nevű változóhoz rendelt **9** értékkel.

7.1.3 Változó argumentumként történő használata

Az előző két példában a tabla() függvény argumentuma mindegyik esetben egy konstans (13 illetve 9) volt. Ez egyáltalán nem kötelező. ***Függvényhívásban használt argumentum változó is lehet***, mint az alábbi példában. Elemezze a példát, próbálja ki és írja le a füzetébe amit kaptott. Magyarázza el a saját szavaival, hogy mi történik ! Ennek a példának az alapján kell, hogy legyen elképzelése arról, hogy milyen hasznosak a függvények az összetett feladatok végzése során :

```
>>> a = 1
>>> while a <20:
...     tabla(a)
...     a = a +1
...
```

Fontos megjegyzés :

A fenti példában a **tabla()** függvénynek átadott argumentum az **a** változó tartalma. A függvény belsejében ez az argumentum hozzá van rendelve az **alap** paraméterhez, ami egy teljesen más változó. Jegyezzük jól meg :

Egy argumentumként átadott változó nevének semmi köze sincs a függvénybeli megfelelő paraméter nevéhez.

Ha akarjuk, ezek a nevek azonosak is lehetnek, de meg kell, hogy értsük, hogy nem ugyanazt a dolgot jelölik (annak ellenére, hogy azonos értéket tartalmazhatnak).

(7) Gyakorlat :

7.1. Importálja a ***turtle*** modult, hogy egyszerű rajzokat tudjon készíteni.

Különböző színű egyenlőoldalú háromszögek sorozatát kell elkészíteni.
Ehhez definiáljon egy **haromszog()** függvényt, ami egy meghatározott színű háromszöget tud rajzolni (ez azt jelenti, hogy a függvény definíciójának egy paramétert kell tartalmazni, ami a szín nevét fogadja).

Alkalmazza ezt a függvényt úgy, hogy ugyanazt a háromszöget különböző helyeken reprodukálja, miközben minden alkalommal megváltoztatja a színt.

7.1.4 Függvény több paraméterrel

A **tabla()** függvény biztosan érdekes, de mindíg csak a szorzótábla első tíz tagját íratja ki, pedig azt is kívánhatjuk, hogy írasson ki más tagokat. Ha csak ez a baj. Úgy javítunk rajta, hogy a **szorzoTabla()** -nak nevezett új verzióban további paraméterekkel egészítjük ki. :

```
>>> def szorzoTabla(alap, kezdete, vege):
...     print 'Az', alap, '-ös szorzótábla :'
...     n = kezdete
...     while n <= vege :
...         print n, 'x', alap, '=', n * alap
...         n = n +1
```

Ez az új függvény három paramétert használ : az előző példabeli szorzótábla alapszámát, az első és az utolsó kiírandó tag indexét.

Próbáljuk ki a függvényt, például :

```
>>> szorzoTabla(8, 13, 17)
```

-t beírva, ami az alábbi kiírást eredményezi :

```
A 8-as szorzótábla részlete :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Megjegyzések :

- Többparaméteres függvény definiálásához elég, ha a paramétereket vesszővel elválasztva beírjuk a zárójelbe.
- A függvény hívásakor az argumentumokat ***ugyanabban a sorrendben*** kell megadni, mint a megfelelő paramétereket (szintén vesszővel elválasztva). Az első argumentum az első paraméterhez, a második argumentum a második paraméterhez lesz rendelve és így tovább.
- Gyakorlatképpen próbálja ki a következő utasítás sorozatot és írja le a füzetébe a kapott eredményt :

```
>>> t, d, f = 11, 5, 10
>>> while t<21:
...     szorzoTabla(t,d,f)
...     t, d, f = t +1, d +3, f +5
...
```

7.2 Lokális változók, globális változók

Ha egy függvénytörzsben definiálunk változókat, akkor ezekhez a változókhoz csak maga a függvény férhet hozzá. Azt mondjuk, hogy ezek a változók a függvényre nézve ***lokális változók***. Ez a helyzet például az előző gyakorlatbeli ***alap, kezdete, vege*** és az ***n*** változók esetében.

Minden alkalommal, amikor a **szorzoTabla()** függvényt hívjuk a Python a számítógép memóriájában ***egy új névteret (namespace)***[24]***-t*** foglal neki. Az **alap, kezdete, vege** és az **n** változók tartalmát ebben a névtérben tárolja, ami a függvényen kívül hozzáférhetetlen. Így például ha rögtön az előző gyakorlat után megpróbáljuk kiíratni az **alap** változó tartalmát, egy hibaüzenetet kapunk :

```
>>> print alap
Traceback (innermost last):
  File "<pyshell#8>", line 1, in ?
    print alap
NameError: alap
```

A gép világosan jelzi, hogy az **alap** szimbólum számára ismeretlen, jóllehet a **szorzoTabla()** függvény korrekten írta azt ki. Az **alap** szimbólumot tartalmazó névtér szigorúan csak a **szorzoTabla()** belső működése számára van fenntartva és automatikusan megszűnik mihelyt a függvény befejezte feladatát.

Függvényen kívül definiált változók a ***globális változók***. Tartalmuk « látható » a függvény belsejéből, de a függvény nem tudja őket módosítani. Példa :

```
>>> def maszk():
...     p = 20
...     print p, q
...
>>> p, q = 15, 38
>>> maszk()
20 38
>>> print p, q
15 38
```

Elemezzük figyelmesen a példát :

Egy nagyon egyszerű függvény definiálásával kezdjük (aminek nincs paramétere).
Ennek a belsejében definiálunk egy **p** változót 20 kezdő értékkel. Mivel ez a **p** változó egy függvény belsejében van definiálva, ezért az egy ***lokális változó.***

A függvény definiálását befejezve visszatérünk a főprogram szintjére, hogy ott két változót **p** -t és **q** -t definiáljuk, amikhez a 15 és 38 tartalmakat rendeljük. Ez a két változó a főprogram szintjén van definiálva, tehát ***globális változók*** lesznek.

Így ugyanazt a változónevet **p** -t kétszer használtuk, ***két különböző változó definiálására*** : az egyik globális, a másik lokális. A gyakorlat során magállapíthatjuk, hogy ez a két változó egymástól különböző, független változó, melyek egy prioritási szabálynak tesznek eleget, ami azt parancsolja, hogy egy függvény belsejében (ahol versenghetnének) a lokálisan definiált változóknak van prioritása.

24 A *névtér (namespace)* fogalmát fokozatosan fogom kifejteni. Ugyancsak a későbbiekben fogjuk megtanulni, hogy a függvények valójában objektumok, amiknek minden egyes hívásakor egy új példányát hozzuk létre.

Megállapítjuk, hogy amikor a **maszk()** függvényt elindítjuk, a **q** globális változó elérhető, mert ezt korrekten írja ki. A **p** változó esetében viszont a lokálisan hozzárendelt értéket írja ki.

Először azt hihetnénk, hogy a **maszk()** függvény egyszerűen módosította a globális **p** változó tartalmát (mivel az hozzáférhető). A következő sorok azt igazolják, hogy ez nem igaz : a **maszk()** függvényen kívül a **p** globális változó megőrzi kezdeti értékét.

Elsőre mindez bonyolultnak tűnhet. Mégis hamar meg fogjuk érteni, hogy mennyire hasznos, hogy a változók egy függvény belsejében lokális változókként vannak definiálva, ami azt jelenti, hogy valamilyen módon be vannak zárva a függvény belsejébe. Valójában ez azt jelenti, hogy a függvényeket mindíg úgy használhatjuk, hogy a legcsekélyebb mértékben sem kell foglalkoznunk a bennük használt változónevek világával : ezek a változók sohasem ütköznek azokkal a változókkal, amiket máshol definiáltunk.

Ezt a helyzetet azonban módosíthatjuk, ha akarjuk. Megtörténhet például, hogy definiálni kell egy függvényt, ami módosítani tud egy globális változót. Ahhoz, hogy ezt elérjük a **global** utasítást kell alkalmazni. Ez az utasítás lehetővé teszi, hogy egy függvénydefiníció belsejében jelezzük, mely változókat kell globális változókként kezelni.

Az alábbi példában a **novel()** függvény belsejében használt **a** változó nemcsak hozzáférhető, hanem módosítható is, mert explicit módon jeleztük, hogy ez egy olyan változó, amit globálisként kell kezelni. Összehasonlításul próbálja ki ugyanezt a gyakorlatot úgy, hogy törli a **global** utasítást : az **a** változó nem inkrementálódik többet a függvény minden egyes hívásakor.

```
>>> def novel():
...  global a
...  a = a+1
...  print a
...
>>> a = 15
>>> novel()
16
>>> novel()
17
>>>
```

7.3 « Igazi » függvények és eljárások

A pontosság kedvéért : az eddig leírt függvények szigorúan véve nem függvények, hanem eljárások[25] (procedurák). Egy (szigorú értelemben vett) « igazi » függvénynek ***egy értéket kell visszaadni***, amikor befejeződik. Egy « igazi » függvényt az olyan kifejezésekben, mint az `y = sin (a)` az *egyenlőségjel* jobboldalán használhatunk. Könnyen belátjuk, hogy ebben a kifejezésben a **sin ()** függvény egy értéket ad vissza (az **a** argumentum sinusát) amit közvetlenül az y változóhoz van rendelve.

Kezdjük egy rendkívül egyszerű példával :

```
>>> def cube(w):
...     return w*w*w
...
```

A **return** utasítás azt definiálja, hogy mi legyen a függvény által visszaadott érték. Jelen esetben a függvényhíváskor átadott argumentum köbéről van szó. Példa :

```
>>> b = cube(9)
>>> print b
729
```

Egy kicsit kidolgozottabb példaként most módosítani fogjuk némileg a **tabla()** függvényt, amin már sokat dolgoztunk azért, hogy egy visszatérési értéket adjon meg. Ez az érték jelen esetben egy lista lesz (a kiválasztott szorzótábla első tíz eleme). Így megint alkalmunk van a listákról beszélni. Ezt arra használom ki, hogy menet közben megtanítok még egy új fogalmat:

```
>>> def tabla(alap):
...     eredmeny = []                  # result először egy üres lista
...     n = 1
...     while n < 11:
...         b = n * alap
...         eredmeny.append(b)      # hozzáad egy tagot a listához
...         n = n +1                    # (lásd a magyarázatott lentebb)
...     return eredmeny
...
```

A függvény teszteléséhez beírhatjuk például :

```
>>> ta9 = tabla(9)
```

Így a ta9 változóhoz lista formájában hozzárendeljük a 9-es szorzótábla első tiz tagját. :

```
>>> print ta9
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print ta9[0]
9
>>> print ta9[3]
36
>>> print ta9[2:5]
[27, 36, 45]
>>>
```

(Ismétlés : egy lista első eleme a 0 indexnek felel meg.)

25 Bizonyos programozási nyelvekben a függvényeket és az eljárásokat különböző utasításokkal definiálják. A Python ugyanazt a **def** utasítást használja mindkettőre.

Megjegyzések:

- Mint azt az előző példában láttuk, a **return** utasítás definiálja, hogy mi legyen a függvény által « visszaadott » érték. Jelen esetben a **result** változó tartalmáról van szó, vagyis a függvény által generált számok listájáról.[26].
- A **result.append(b)** utasítás a második példánk egy fontos fogalom alkalmazására, amire később még hosszabban visszatérünk : *ebben az utasításban az append()* ***metódust*** *alkalmazzuk a result* ***objektumra****.*

  Fokozatosan fogjuk pontosítani, hogy mit kell érteni a programozásban ***objekum*** alatt. Most fogadjuk el, hogy ez egy nagyon általános fogalom, ami a Python listákra is ráillik.
  Egy ***metódus*** valójában nem más mint egy függvény (amit egyébként a zárójelek jelenlétéről ismerhetünk fel), de egy olyan függvény, ami egy objektumhoz van kötve.
  Részét képezi ezen objektum definíciójának, pontosabban azon speciális ***osztály*** (***class***) definíciójának, amihez ez az objektum tartozik (a ***class*** fogalmát később fogjuk tanulni).

  Egy objektumhoz kapcsolódó metódus használata ezen objektum valamilyen speciális módon történő « működtetéséből » áll. Például az **objektum3** objektum **metodus4()** metódusát egy **objektum3.metodus4()** típusú utasítással használjuk, vagyis az objektum nevét követi a metódus neve, a kettőt pont köti össze. Ez a pont fontos szerepet játszik : egy igazi ***operátornak*** tekinthetjük.

  Példánkban az **append()** metódust alkalmazzuk a **result** objektumra. A Pythonban a listák egy speciális objektumtípust alkotnak, amire egy egész sor metódus alkalmazható. Jelen esetben az **append()** metódus a listák egy speciális függvénye, ami arra való, hogy a lista végéhez hozzákapcsoljunk egy elemet. A hozzákapcsolandó elemet, mint minden argumentumot, zárójelben adjuk meg a metódusnak.

  Megjegyzés :
  Hasonló eredményt kaphattunk volna, ha e helyett az utasítás helyett a « **result = result + [b]** » kifejezést használtuk volna. Ez az eljárás kevésbé ésszerű és hatékony, mert a ciklus minden lefutásakor egy ***új*** **result** listát kell definiálni, amibe minden alkalommal, amikor egy kiegészítő elemet kapcsolunk a listához, az egész előző listát be kell másolni.
  Viszont az **append()** metódus használatakor a számítógép egy már létező lista módosításához fog hozzá (anélkül, hogy azt egy új változóba másolná át). Ez az ajánlott technika, mert ez kevésbé terheli le a számítógép erőforrásait (különösen ha nagyméretű listákról van szó).

- Egyáltalán nem szükséges, hogy egy függvény visszatérési értéke hozzá legyen rendelve egy változóhoz (ahogyan azt eddig az érthetőség kedvéért tettük). Így az alábbi parancsokat beírva tesztelhettük volna a cube() és tabla() függvényeket :

```
>>> print cube(9)
>>> print tabla(9)
>>> print tabla(9)[3]
```

vagy még egyszerűbben :

```
>>> cube(9)
```

... stb.

26 A **return**-t argumentum nélkül is alkalmazhatjuk egy függvény belsejében, hogy előidézzük az azonnali visszatérést a hívó programba. Ebben az esetben a visszatérési érték a **None** objektum (egy speciális objektum, ami a «"semmi"-nek felel meg).

7.4 Függvények használata scriptben

A függvények ezen első közelítéséhez eddig csak a Python interpreter interaktív módját használtuk.

Nyilvánvaló, hogy a függvények scriptekben is használhatók. Próbáljuk ki az alábbi programot, ami egy gömb térfogatát számolja ki az ismert formula segítségével : $V = \frac{4}{3}\pi R^3$

```
def cube(n):
   return n**3

def gombTerfogat(r):
   return 4 * 3.1416 * cube(r) / 3

r = input('Írja be a sugár értékét : ')
print 'A gömb térfogata :', gombTerfogat(r)
```

Megjegyzések :

Nézzük jól meg, a programnak három része van : a két függvény: **cube()** és **gombTerfogat()**, valamint a program törzse.

A program törzsében van egy függvényhívás: **gombTerfogat()**.

A **gombTerfogat()** függvény belsejében van a **cube()** függvényhívás.

Figyeljük meg, hogy a program három része bizonyos sorrendben van elrendezve : először a függvénydefiníciók jönnek, majd a program törzse következik. Azért kell ez az elrendezés, mert az interpreter a program utasítássorait egymás után abban a sorrendben hajtja végre, ahogyan azok a forráskódban megjelennek. Tehát a scriptben a függvénydefinícióknak meg kell előzni a függvények használatát.

Fordítsuk meg ezt a sorrendet, hogy meggyőződjünk erről (tegyük előre a program törzsét), és jegyezzük meg az ilymódon módosított script végrehajtásakor kiírt hibaüzenet típusát.

Valójában egy Python programnak a törzse egy speciális entitást képez, amit az interpreter a belső működése során mindíg a **__main__** foglalt szó alatt ismer fel (a « *main* » jelentése : fő. Kettős aláhúzás karakter veszi köröl, hogy elkerüljük más szimbólumokkal való összetévesztését.) Egy script végrehajtása mindíg ennek a ***__main__*** entitásnak az első utasításával kezdődik, bárhol is legyen az a listában. A következő utasításokat egymás után sorban hajtja végre az interpreter egészen az első függvényhívásig. Egy függvényhívás olyan, mint egy kerülő a végrehajtás menetében : a következő utasításra való áttérés helyett az interpreter a hívott függvényt hajtja végre, majd visszatér a hívó programba, hogy folytassa a megszakított munkát. Hogy ez a mechanizmus működni tudjon, az kell, hogy az interpreter a **__main__** entitás ***előtt*** el tudja olvasni a függvénydefiníciót, tehát a **__main__** általában a script végére van téve.

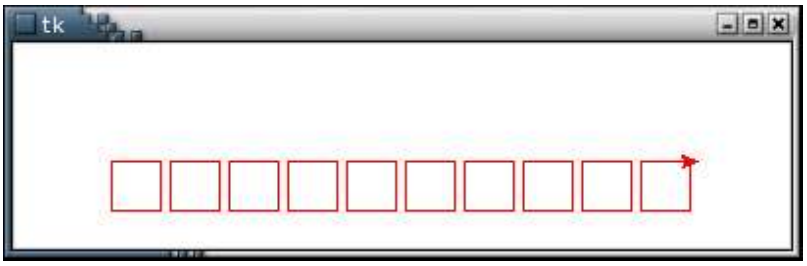
Példánkban a **__main__** entitás hív egy függvényt, ami egy másik függvényt hív. Ez egy nagyon gyakori helyzet a programozásban. Ha elfogadható módon szeretnénk megérteni, hogy mi történik egy programban, akkor meg kell tanulnunk scriptet olvasni, de nem az első sortól az utolsóig, hanem inkább egy olyan útvonalat követve, mint amin a script végrehajtása történik. Ez konkrétan azt jelenti, hogy gyakran az utolsó soraival kell elkezdeni egy script elemzését !

7.5 Függvénymodulok

Annak érdekében, hogy még jobban megértsük egy függvény definíciója és programbeli használata között a különbséget, azt ajánlom, hogy gyakran helyezzük el a függvénydefinícióinkat egy Python modulban és az azokat használó programot egy másik modulban.

Példa :

A *turtle* modul segítségével szeretnénk az alábbi rajzokat elkészíteni *:*



Írja be és mentse el egy ***teknos_rajz.py*** nevű file-ba a következő kódsorokat :

```
from turtle import *

def negyzet(meret, szin):
    "meghatározott méretű és színű négyzetet rajzoló függvény"
    color(szin)
    c =0
    while c <4:
        forward(meret)
        right(90)
        c = c +1
```

Bizonyára észrevette, hogy a negyzet() függvény egy karakterlánccal kezdődik. Ennek a stringnek semmilyen funkcionális szerepe sincs a scriptben : a Python úgy kezeli, mint egy egyszerű kommentet, azonban ez automatikusan külön el van tárolva egy belső dokumentációs rendszerben, amit azután egyes felhasználói programok és "intelligens" szövegszerkesztők használhatnak.

Ha például az IDLE környezetben dolgozunk, amikor ílymódon dokumentált függvényeket hívunk, mindíg egy help-buborékban fogjuk látni feltűnni ezt a dokumentációs stringet.

A Python egy __doc__ nevű (a "doc" szót kettős "_" karakter veszi köröl) speciális változóban helyezi el ezt a stringet, ami a függvényobjektumhoz van kötve, lévén annak egyik attribútuma. (Amikor rátérünk az objektumok osztályaira, többet fogunk erről tanulni. Lásd 155. oldalon).

Így bármely függvény dokumentációs stringjét ennek a változónak a kiíratásával találhatjuk meg. Példa :

```
   >>> def proba():
...     "Ez a függvény jól van dokumentálva, de szinte semmit sem csinál."
...     print "semmi jeleznivaló sincs"

>>> proba()
semmi jeleznivaló sincs

>>> print proba.__doc__
Ez a függvény jól van dokumentálva, de szinte semmit sem csinál.
```

Vegyük a fáradtságot és a jövőben valamennyi függvénydefiníciónkban helyezzünk el ilyen magyarázó stringeket : ez egy követendő gyakorlat.

Az általunk létrehozott file mostantól fogva egy valódi Python függvénymodul ugyanúgy, mint a már ismert *turtle* és *math* modulok. Így bármilyen scriptben használhatjuk, mint például ebben, amelyik a kért rajzolást elvégzi :

```
from teknos_rajz import *

up()                    # fölemeli a ceruzát
goto(-150, 50)          # balra felmegy

# tíz piros, sorbarendezett négyzetet rajzol:
i = 0
while i < 10:
    down()              # leteszi a ceruzát
    carre(25, 'red')    # egy négyzetet rajzol
    up()                # fölemeli a ceruzát
    forward(30)         # távolabb megy
    i = i +1

a = input()             # vár
```

Megjegyzés :

A priori úgy nevezhetjük el a függvénymoduljainkat, ahogy tetszik. De tudnunk kell, hogy lehetetlen importálni egy modult, ha a neve a 23. oldalon megadott 29 foglalt Python szó egyike; mert az importált modul neve a scriptünkben egy változónévvé válna, és a foglalt szavakat nem lehet változónevekként használni. Arra is emlékezzünk vissza, hogy kerülni kell, hogy egy már létező Python modul nevét adjuk a moduljainknak és általában a scriptjeinknek, ellenkező esetben ütközésekre kell számítanunk. Például ha a ***turtle.py*** *nevet adjuk egy gyakorlatnak, amiben elhelyeztük a* ***turtle*** *modult importáló utasítást, akkor magát a gyakorlatot fogjuk importálni !*

Egy Python program struktúrája

```
####################################
#Python program                    #
#Szerző :G. Swinnen, Liége, 2003   #
#Licenc : GPL                      #
####################################


####################################
#Külső függvények importálása      #

from math import sqrt


####################################
#Függvények lokális definíciója    #
from math import sqrt

def elofordulas(car, ch):
    "karakterek száma - <car> \
     karakterlánc - <ch>"

    nc = 0
    i = 0
    while i < len(ch):
        if ch[i]== car:
            nc = nc + 1
        i = i + 1

return nc


####################################
#Főprogram törzse:                 #

print "Írjon be egy számot:"
nbr = input

print "Írjon be egy mondatot:"
phr = raw_input()
print "A megszámolandó karakter:"
cch = raw_input()

no = elofordulas(cchr, phr)
rc = sqrt(nbr**3)

print "A szám köbének",
print "négyzetgyöke",
        i = i + 1
print rc
print "A mondat", no, "számú",
print cch, "karaktert tartalmaz"
```

Egy Python program általában a következő blokkokat tartalmazza sorrendben:
- *Inicializáló utasítások (függvények és/vagy osztályok importálása*
- *Lokális függvény- és/vagy osztálydefiníciók*
- *A főprogram törzse*

A program tetszőleges számú lokálisan definiált vagy külső modulokból importált függvényt használhat. (Mi magunk is definiálhatunk ilyen modulokat.)

Egy függvénydefiníció gyakran tartalmaz paraméterlistát : a paraméterek mindig változók, amik a függvény hívásakor kapnak értéket

Egy 'while' típusú programhuroknak elvben a következő 4 elemet kell tartalmazni:
- *egy számláló inicializálását*
- *a while utasítást, amiben megadjuk a while-t követő utasítások megismétlésének feltételét*
- *a megismétlendő utasításblokkot*
- *a számlálót inkrementáló utasítást*

A függvény mindig egy jól meghatározott értéket 'ad vissza' a hívó programnak.
(Ha nem használjuk a return utasítást vagy ha argumentum nélkül használjuk, akkor a függvény visszatérési értéke egy üres objektum : <None>

A függvényt hívó program szokás szerint argumentumokat ad át a függvénynek, melyek értékek, változók vagy kifejezések lehetnek

Exercices :

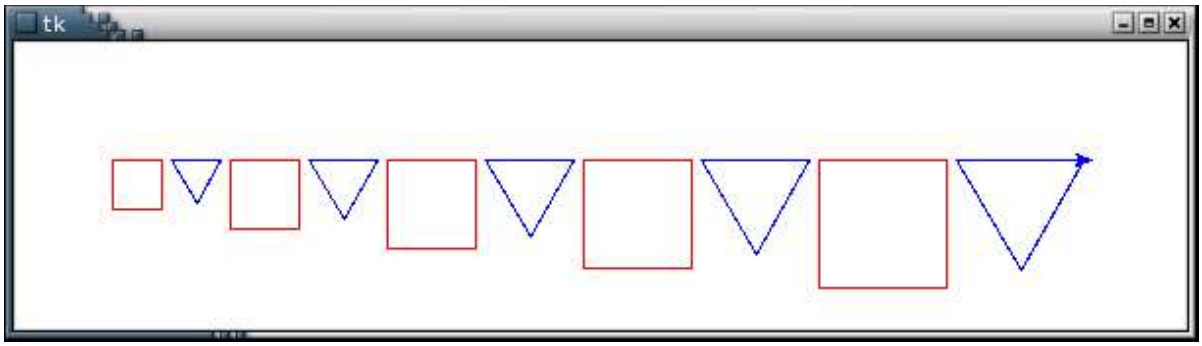
7.2. Definiáljon egy **karakterSor(n, ca)** függvényt, ami **n** darab **ca** karakterből álló stringet ad vissza.

7.3. Definiáljon egy **korTerulet(R)** függvényt. **A** függvénynek egy kör területét kell visszaadni, aminek az R sugarát argumentumként adjuk meg.
Például a **:**`print korTerulet(2.5)` utasítás eredményének **19.635** -nek kell lenni

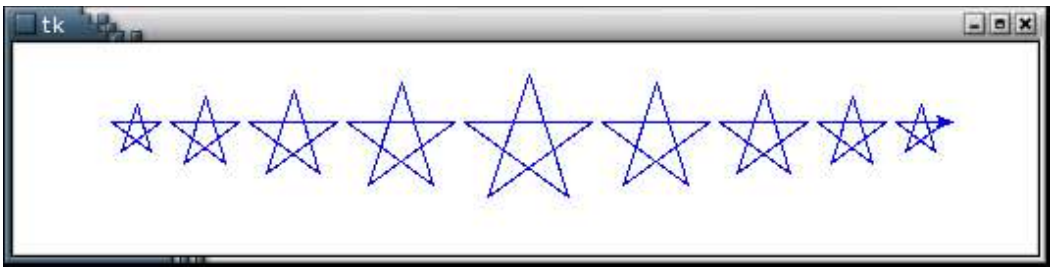
7.4. Definiáljon egy **dobozTerfogat(x1,x2,x3)** függvényt, ami egy parallelepipedon térfogatát adja vissza, aminek a méreteit az **x1**, **x2**, **x3** argumentumokban adjuk meg.
Például a : `print dobozTerfogat((5.2, 7.7, 3.3)` utasítás végrehajtásakor eredményül **132.13** -at kell kapnunk.

7.5. Definiáljon egy **maximum(n1,n2,n3) f**üggvényt, ami az argumentumként megadott n1, n2, n3 számok közül a legnagyobbat adja vissza. Például a : `print maximum(2,5,4)` utasítás végrehajtásakor eredményül **5** -öt kell kapnunk.

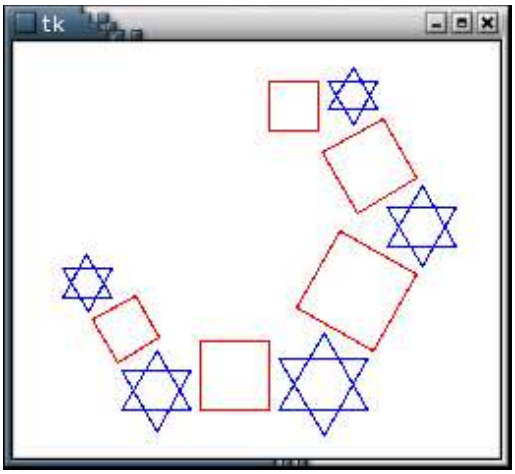
7.6. Fejezze be a 73. oldalon leírt ***teknos_rajz.py*** grafikus függvénymodult.
Adjon úgy egy **szog** paramétert a **negyzet()** függvényhez, hogy a négyzeteket különböző orientációkban lehessen kirajzolni.
Utána definiáljon egy **haromszog(meret, szin, szog)** függvényt, ami meghatározott méretű, színű, orientációjú egyenlő oldalú háromszöget tud rajzolni.
Ellenőrizze a modult egy program segítségével, ami különböző argumentumokkal fogja hívni ezeket a függvényeket, hogy egy sorozat négyzetet és háromszöget rajzoljon :

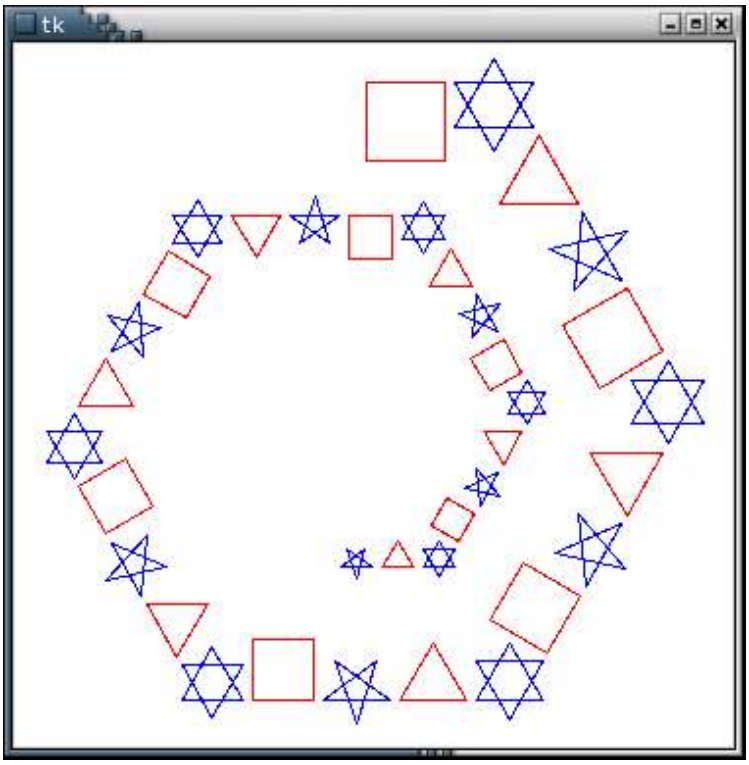


7.7. Adjon az előző gyakorlat moduljához egy **csillag5()** függvényt, ami egy ötágú csillagot tud rajzolni. A főprogramba illesszen be egy ciklust, ami vízszintesen 9 különböző méretű kis csillagot rajzol :



7.8. Adjon az előző gyakorlat moduljához egy **csillag6()** függvényt, ami egy hatágú csillagot tud rajzolni. Ez két egymásra rajzolt egyenőoldalú háromszögből áll. Ennek az új függvénynek az előzőleg definiált **haromszog()** függvényt kell hívni. A főprogram egy csillagsorozatot fog rajzolni :





7.9. D**efiniáljon egy `karakterSzam(ca,ch)`** függvényt, ami visszaadja a **ch** stringben előforduló **ca** karakterek számát.
`Például a :` **`print karakterSzam('e','Cette phrase est un exemple')`** utasítás végrehajtásakor eredményül 7 -et kell kapnunk.

7.10. Definiáljon egy **indexMax(lista)** függvényt, ami az argumentumként megadott lista legnagyobb értékű elemének az indexét adja vissza.
Alkalmazási példa :

```
sorozat = [5, 8, 2, 1, 9, 3, 6, 7]
print indexMax(sorozat)
```

```
4
```

7.11. Definiáljon egy **honapNeve(n)** függvényt, ami az év n-edik hónapjának nevét adja vissza. Például a : `print honapNeve(4)` utasításnak az **`Április`** eredményt kell adni.

7.12. Definiáljon egy **inverse(ch)** függvényt, ami bármilyen stringben megfordítja a karakterek sorrendjét. (Az invertált stringet fogja visszaadni a hívó programnak.)

7.13. Definiáljon egy **szavakSzama(mon)** függvényt, ami a **mon** mondatban található szavak számát adja vissza. (Szavaknak olyan karaktercsoportokat tekintünk, amik között betűköz van).

7.6 Paraméterek típusadása

Megtanultuk, hogy a típusadás a Pythonban dinamikus. Ez azt jelenti, hogy egy változónak a típusa abban a pillanatban van definiálva, amikor hozzárendelünk egy értéket. Ez a mechanizmus működik a függvényparaméterekre is. A paraméter típusa meg fog egyezni a függvénynek átadott argumentum típusával. Példa :

```
>>> def kiir3szor(arg):
...     print arg, arg, arg
...

>>> kiir3szor(5)
5 5 5

>>> kiir3szor('zut')
zut zut zut

>>> kiir3szor([5, 7])
[5, 7] [5, 7] [5, 7]

>>> kiir3szor(6**2)
36 36 36
```

Ebben a példában megállapíthatjuk, hogy ugyanaz a **kiir3szor()** függvény mindegyik esetben elfogadja az átadott argumentumot, legyen az egy szám, egy string, egy lista vagy éppen egy kifejezés. Ez utóbbi esetben a Python a kifejezés kiértékelésével kezdi és az eredményt adja át argumentumként a függvénynek.

7.7 Alapértelmezett értékek adása a paramétereknek

Egy függvénydefinícióban lehetséges (és gyakran kívánatos) mindegyik paraméternek egy alapértelmezett értéket definiálni. ***Így egy olyan függvényt kapunk, ami úgy is hívható, hogy a várt argumentumoknak csak egy részét adjuk meg***. Példák :

```
>>> def udvarias(nev, megszolitas ='Úr'):
...    print "Fogadja ", nev, " ", megszolitas, " jókívánságaimat."
...

>>> udvarias('Dupont')
Fogadja Dupont Úr jókívánságaimat.

>>> udvarias('Durand', 'Kisasszony')
Fogadja Durand Kisasszony jókívánságaimat.
```

Ha ezt a függvényt csak az első argumentuma megadásával hívjuk, akkor a második egy alapértelmezett értéket vesz fel. Ha mindkét argumentumot megadjuk, akkor a második argumentum alapértelmezett értékét figyelmen kívül hagyja.

Definiálhatunk minden paraméternek, vagy csak egy részüknek alapértelmezett értéket. Ebben az esetben azonban az alapértelmezett érték nélküli paramétereknek meg kell a listában előzni a többieket. Például az alábbi definíció inkorrekt :

```
>>> def udvarias( megszolitas='Úr', nev):
```

Másik példa :

```
>>> def kerdes(uzenet, kiserlet =4, kerem ='Igen vagy nem ?'):
...     while kiserlet >0:
...         valasz = raw_input(uzenet)
...         if valasz in ('i', 'igen','I','Igen','IGEN'):
...             return 1
...         if valasz in ('n','nem','N','Nem','NEM'):
...             return 0
...         print kerem
...         kiserlet = kiserlet-1
...
>>>
```

Ezt a függvényt különböző módokon lehet hívni, például :

```
rep = kerdes('Valóban be akarja fejezni ? ')
```

vagy :

```
rep = kerdes('Törölni kell ezt a filet ? ', 3)
```

vagy :

```
rep = kerdes('Megértette ? ', 2, 'Válaszoljon igennel vagy nemmel !')
```

(Vegye a fáradságot és elemezze ezt a példát !)

7.8 Argumentumok címkével

A programozási nyelvek többségében a függvényhíváskor megadott argumentumokat ***pontosan ugyanabban a sorrendben*** kell megadni, mint ami a függvénydefinícióban nekik megfelelő paraméterek sorrendje.

A Python viszont jóval nagyobb rugalmasságot enged meg. Ha a függvénydefinícióban a paraméterek mindegyike a fentebb leírt formában kap egy alapértelmezett értéket, akkor hívhatjuk úgy a függvényt, hogy az argumentumait tetszőleges sorrendben adjuk meg, feltéve, hogy a megfelelő paramétereket névszerint adjuk meg. Példa :

```
>>> def madar(volts=100, state='lefagyott', action='java-t táncolni')
...     print 'A papagáj nem tudna ', action
...     print 'ha rákapcsolná a', volts, 'V-ot !'
...     print "E mondóka szerzője teljesen", state
...

>>> madar(state='tűzbejött', volts=250, action='helyeselni')
A papagáj nem tudna helyeselni
ha rákapcsolná a 250 V-ot !
A szerző teljesen tűzbejött

>>> madar()
A papagáj nem tudna java-t táncolni
ha rákapcsolná a 100 V-ot !
A szerző teljesen lefagyott
```

Gyakorlatok :

7.14. Módosítsa az egyik előző példában definiált **dobozTerfogat(x1,x2,x3)** függvényt úgy, hogy 3, 2, 1 argumentummal, vagy akár argumentum nélkül is lehessen hívni. Alapértelmezésben az argumentum értéke legyen 10.
Például :

```
print dobozTerfogat()          az eredmény :  1000
print dobozTerfogat(5.2)       az eredmény  :  250
print dobozTerfogat(5.2, 3)    az eredmény  :  156.0
```

7.15. Módosítsa a fenti **dobozTerfogat(x1,x2,x3)** függvényt olymódon, hogy 3, 2 vagy 1 argumentummal lehessen hívni. Ha egy argumentumot használunk, akkor a dobozt kockának tekintjük (az argumentum a kocka oldala). Ha két argumentumot használunk, akkor a dobozt négyzet alapú prizmának tekintjük. (Ebben az esetben az első argumentum a négyzet oldala és a második a prizma magassága). Ha három argumentumot használunk, akkor a dobozt parallelepipedonnak tekintjük.
Például :

```
print dobozTerfogat()          az eredmény : -1       (→ hibajelzés)
print dobozTerfogat(5.2)       az eredmény : 140.608
print dobozTerfogat(5.2, 3)    az eredmény : 81.12
print dobozTerfogat(5.2, 3, 7.4)  az eredmény : 115.44
```

7.16. Definiálja a **karakterCsere(ch,ca1,ca2,kezdo,vegso)** függvényt, ami a **ch** karakterláncban valamennyi **ca1** karaktert a **kezdo** indextől kezdve a **vegso** indexig a **ca2** karakterrel helyettesíti, az utolsó két argumentum elhagyható (ebben az esetben a karakterlánc elejétől a végéig helyettesítendő a ca1 karakter.) Példák a végrehajtásra :

```
>>> mondat = 'Ceci est une toute petite phrase.'
>>> print karakterCsere(mondat, ' ', '*')
Ceci*est*une*toute*petite*phrase.
>>> print karakterCsere(mondat, ' ', '*', 8, 12)
Ceci est*une*toute petite phrase.
>>> print karakterCsere(mondat, ' ', '*', 12)
Ceci est une*toute*petite*phrase.
>>> print karakterCsere(mondat, ' ', '*', vegso = 12)
Ceci*est*une*toute petite phrase.
```

7.17. Definiálja az **maxElem(lista,kezdo,vegso)** függvényt, ami a paraméterként átadott **lista** listából megadja legnagyobb értékű elemet. A **kezdo** és **vegso** argumentumok adják meg azt a két indexet, melyek között végre kell hajtani a keresést. Közülük bármelyik elhagyható (az előző gyakorlathoz hasonlóan). Példa a várt függvényvégrehajtásra :

```
>>> serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
>>> print maxElem(serie)
9
>>> print maxElem(serie, 2, 5)
7
>>> print maxElem(serie, 2)
8
>>> print maxElem(serie, vegso =3, kezdo =1)
6
```

8. Fejezet : Az ablakok és a grafika használata

Eddig a Pythont kizárólag « szövegmódban » használtuk. Azért jártunk így el, mert tisztázni kellett néhány elemi fogalmat és a nyelv alapszerkezetét, mielőtt komplexebb objektumokat (ablakok, képek, hangok, stb.) igényelő kísérleteket tervezünk. Most megengedhetünk magunknak egy kis kiruccanást a grafikus interface-ek világába. Ez csak ízelítő lesz. Még sok alapvető dolog van, amit meg kell tanulni és közülük soknak a szöveges megközelítés a legmegfelelőbb.

8.1 Grafikus interface-ek (GUI)

Ha az olvasó még nem tudja, úgy most tapasztalni fogja, hogy grafikus interface-ek (vagy *GUI : Graphical User Interface*) területe rendkívül komplex. Minden operációs rendszer több grafikus alap « függvénykönyvtárat » kínálhat, amihez gyakran társul még számos kiegészítő, többé-kevésbé speciális programozási nyelv. Általában ezek a komponensek objektum-osztályokként jelennek meg, amiknek az attribútumait és metódusait kell majd tanulmányozni.

A Pythonnal leginkább a Tkinter grafikus könyvtárat használják, ami a - kezdetben a Tcl nyelv számára fejlesztett - Tk könyvtár egy adaptációja. Sok más igen érdekes grafikus könyvtárat ajánlottak a Pythonhoz : wxPython, pyQT, pyGTK, stb. Lehetőség van a Java widget-ek és a Windows MFC -k használatára is.

Ebben a jegyzetben a *Tkinter* -re fogunk szorítkozni, aminek szerencsére hasonló (és ingyenes) verziói vannak *Linux*, *Windows* és *Mac* platformokra.

8.2 Első lépések a Tkinter-rel

A magyarázathoz feltételezem, hogy a *Tkinter* modul már telepítve van. Ahhoz, hogy a funkcióit használni tudjuk, a script valamelyik első sorának egy import utasítást kell tartalmazni :

```
from Tkinter import *
```

A Pythonban nem mindíg szükséges scriptet írni. Interaktív módban elindítva számos kísérletet végezhetünk közvetlenül parancssorban. A következő példában egy egyszerű ablakot hozunk létre és két tipikus *widget*[27] -et kapcsolunk hozzá : egy címkét (*label*) és egy nyomógombot (*button*).


```
>>> from Tkinter import *
>>> abl1 = Tk()
>>> tex1 = Label(abl1, text='Jó napot kívánok!', fg='red')
>>> tex1.pack()
>>> gomb1 = Button(abl1, text='Kilép', command = abl1.destroy)
>>> gomb1.pack()
>>> abl1.mainloop()
```

27 A "*widget*" a "*window gadget*" kifejezés összevonásából ered. Bizonyos programozói környezetekben ezeket inkább "control"-nak vagy "grafikus komponens"-nek nevezik. Így nevezik az alkalmazásablakokban elhelyezhető objektumokat, amik például nyomógombok, checkboxpk, képek, stb.lehetnek, és néha magát az ablakot is.

Megjegyzés : A Python verziójától függően az alkalmazásablakot vagy közvetlenül a második sor, vagy csak a hetedik sor bevitele után látjuk majd megjelenni.[28].

Most vizsgáljuk meg részletesebben a parancssorokat :

1. Mint már említettem, könnyű olyan Python *modulokat* létrehozni, amik scripteket, függvénydefiníciókat, objektumosztályokat, stb. tartalmaznak. Ezeket a modulokat vagy teljes egészükben, vagy részben bármely programba, vagy magába az interaktív módban működő interpreterbe (vagyis közvetlenül a parancssorba) importálhatjuk. Ezt tesszük példánk első sorában : « `from Tkinter import *` » importálja a *Tkinter* modulban lévő valamennyi osztályt.

   Egyre gyakrabban kell ***osztályokról*** (***class***-okról) beszélnünk. A programozásban így nevezzük az objektum generátorokat, amik újra felhasználható programrészletek.
   Nem teszek kísérletet az objektumok és az osztályok végleges és precíz definíciójának megadására, inkább azt ajánlom, hogy használjunk néhányat. Menet közben lépésről-lépésre fogjuk finomítani a meghatározásukat.

2. Példánk második sorában : « **abl1** `= Tk()` » -ban, a *Tkinter* modul **Tk()** osztályát használjuk és annak egy ***példányát*** (egy ***objektumát***) hozzuk létre az **abl1** -et.

   Egy ***objektum létrehozása egy osztályból*** a mai programozási technikákban - melyek egyre gyakrabban folyamodnak az « *objektum orientált programozás* »-nak *(vagy OOP : Object Oriented Programming)* nevezett módszerhez - alapvető művelet.

   Az ***osztály*** egy általános model (vagy minta), amiből a gép kérésünkre egy speciális számítástechnikai objektumot hoz létre. Definíciókat és különböző opciókat tartalmaz, amiknek csak egy részét használjuk a belőle létrehozott objektumban. Így a *Tkinter* könyvtár egyik legalapvetőbb osztálya, a **Tk()** osztály mindent tartalmaz, ami különböző típusú, méretű és színű, menüsoros vagy anélküli, stb. alkalmazásablakok létrehozásához szükséges.

   Ezt használjuk fel a grafikus alapobjektumunk, egy ablak létrehozására, ami a többi objektumot fogja tartalmazni. A **Tk()** zárójelében különböző opciókat adhatnánk meg, de ezt egy kicsit későbbre hagyom.

   Az objektumot létrehozó utasítás egy egyszerű értékadáshoz hasonlít. Azonban itt két dolog történik egyszerre :

   - ***egy új objektum jön létre***, (ami összetett lehet és ezért tekintélyes mennyiségű memóriát foglalhat le)
   - ***értéket adunk egy változónak,*** ami ettől fogva az objektum hivatkozásként fog szolgálni[29].

28 Ha *Windows* alatt végezzük ezt a gyakorlatot, akkor azt ajánlom, hogy inkább DOS-ablakban, vagy IDLE-ben használjunk standard Pythont, mint *PythonWin* -t. Jobban megfigyelhetjük, hogy mi történik az egyes utasítások beírása után.

29 A nyelvnek ez a tömörsége annak az egyik következménye, hogy a Pythonban a változók típusadása dinamikus. Más nyelvek speciális utasítást (mint amilyen a **new**) használnak egy új objektum létrehozására.
Példa : **enAutom = new Cadillac** (a **Cadillac** classe egy objektumát hozzuk létre, amire az **enAutom** válozóval hivatkozunk.)

3. A harmadik sorban : « **tex1 = Label(abl1, text='Jó napot !', fg='red')** » egy másik objektumot (egy *widget*-et) hozunk létre, ez alkalommal a **Label()** osztályból.
   Amint a neve is jelzi, ez az osztály mindenféle *címkét* definiál. Ezek szövegtöredékek, amik arra használhatók, hogy különböző információkat írjunk az ablak belsejébe.
   A korrekt kifejezésre törekedve azt fogjuk mondani, hogy a **tex1** objektumot a **Label()** osztályból hoztuk létre (***instanciáltuk vagy példányosítottuk***).

   Itt jegyzem meg, hogy ugyanúgy hívunk egy osztályt, mint egy függvényt : vagyis zárójelben argumentumokat adunk meg. Majd meglátjuk, hogy egy osztály egyfajta konténer, amiben függvények és adatok vannak összegyűjtve.

   Milyen argumentumokat adtunk meg ehhez az instanciáláshoz ?

   - Az első argumentum (**abl1**) azt adja meg, hogy a most létrehozott új *widget*-et ***egy már létező másik widget fogja tartalmazni***. Az utóbbit az új *widget* « master » *widgetjének* nevezzük, az **abl1** *widget* a **tex1** objektum tulajdonosa. (Azt is mondhatjuk, hogy a **tex1** objektum az **abl1** objektum ***slave*** *widget* -je).
   - A következő két argumentum a *widget* pontos alakjának meghatározására való. Ez két ***kezdőérték,*** mindegyikük string formájában van megadva : első a címke szövege, második az előtér (f*oreground* rövidítve **fg**) színe. Esetünkben a kiíratni kívánt szöveg színét pirosnak definiáltuk.
     Egyéb jellemzőket is meg tudnánk még határozni, például : a betűtípust, vagy a háttér színét. Ezeknek a jellemzőknek azonban van egy alapértelmezett értékük a **Label()** osztály belső definícióiban. Csak azoknak a jellemzőknek kell értéket adnunk, melyeket a standard modeltől eltérőnek akarunk.

4. A negyedik sorban : « **tex1.pack()** », a **tex1** objektumhoz tartozó **pack()** ***metódust*** aktiváljuk. Már találkoztunk a *metódus* kifejezéssel (a listáknál). A metódus egy objektumba beépített (azt is fogjuk mondani, hogy egy objektumba ***bezárt***) függvény. Nemsokára megtanuljuk, hogy egy objektum valójában egy programkód részlete, ami mindíg tartalmaz :
   - különböző típusú változókban tárolt adatokat (numerikusokat vagy másféléket) : ezeket az objektum ***attribútumainak*** (vagy tulajdonságainak nevezzük).
   - eljárásokat vagy függvényeket (ezek tehát algoritmusok) : ezeket az objektum ***metódusainak*** nevezzük.

   A **pack()** metódus része egy metóduscsoportnak, aminek a tagjai nemcsak a **Label()** osztály *widget* -jeire alkalmazhatók, hanem más *Tkinter widget*-ekre is. A *widgetek* ablakbeli geometriai elrendezésére hatnak. Ha egyesével írjuk be a példa parancsait, meggyőződhetünk róla, hogy a **pack()** metódus automatikusan akkorára csökkenti a « master » ablak méretét, hogy az elég nagy legyen az előre definiált « slave » *widget* -ek tartalmazásához.

5. Az ötödik sorban : « **gomb1 = Button(abl1, text='Kilép', command = abl1.destroy)** » létrehozzuk második « slave » w*idget* -ünket : egy nyomógombot.
   Ahogyan az előző widget esetén tettünk, a **Button()** class-t hívjuk zárójelek között a paramétereket megadva. Mivel ez alkalommal egy interaktív objektumról van szó, ezért a **command** opcióval meg kell adnunk, hogy mi történjen, amikor a felhasználó a nyomógombra kattint. Ebben az esetben az **abl1** objektumhoz tartozó **destroy** metódust hozzuk működésbe, ami törli az ablakot.

6. A 6. sorban a **pack()** metódus az ablak geometriáját az imént beépített új objektumhoz igazítja.

7. A hetedik sor : « **abl1.mainloop()** » nagyon fontos, mert ez indítja el az ablakhoz kapcsolt ***eseményfogadót,*** ami arra kell, hogy az alkalmazásunk figyelje az egérkattintásokat, billentyűleütéseket, stb. Ez az az utasítás, amelyik valamilyen módon elindítja az alkalmazást.

   Mint a neve (*mainloop* = *főhurok*) mutatja, az **abl1** objektum egyik metódusáról van szó, ami egy programhurkot aktivál. Ez a programhurok háttérprogramként folyamatosan működik és várja az operációs rendszer által küldött üzeneteket. Folyamatosan lekérdezi a környezetét :a bemeneti perifériákat (egér, billentyűzet, stb.). Valamilyen esemény detektálásakor az illető eseményt leíró üzeneteket küld azoknak a programoknak, amiknek szükségük van arra, hogy tudjanak az illető esemény bekövetkezéséről. Nézzük meg ezt egy kicsit részletesebben.

8.3 Eseményvezérelt programok

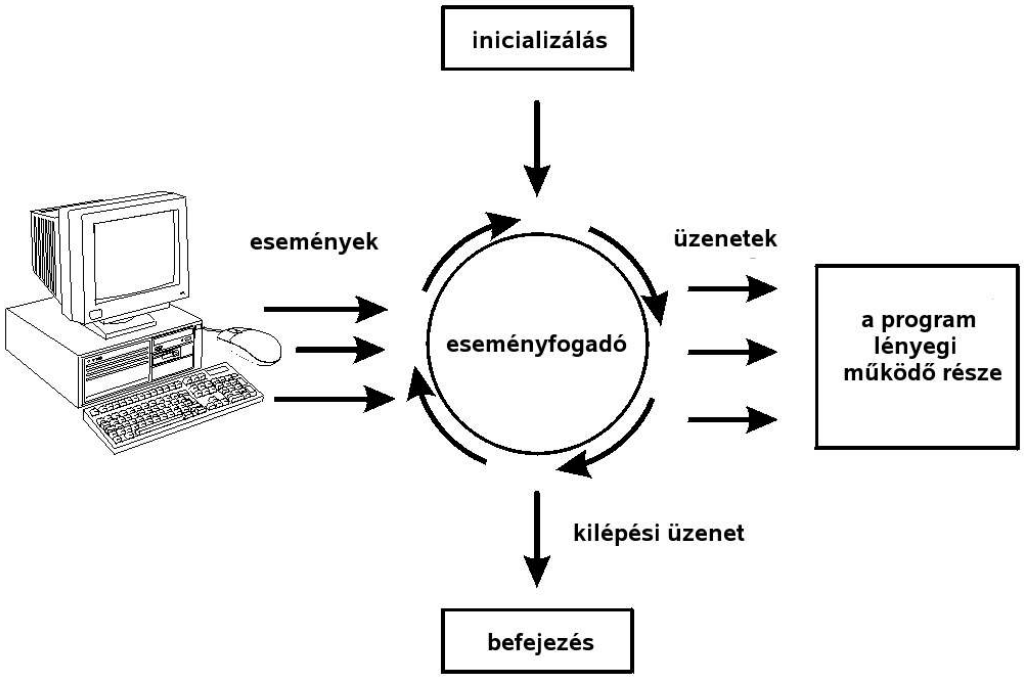
Az első grafikus interface-ű programunkkal kísérleteztünk az imént. Az ilyen típusú program máshogyan van struktúrálva, mint az előzőekben tanulmányozott « szöveges » scriptek.

Minden számítógépprogram működésének nagyjából három fő szakasza van : ***inicializálási szakasz*** : ez az elvégzendő munkára felkészítő utasításokat tartalmazza (a szükséges külső modulok hívása, file-ok megnyitása, adatbázis serverhez vagy az internethez kapcsolódás, stb.); ***lényegi működési szakasz*** : itt összpontosul a lényegi működés (vagyis mindaz, amit a program tesz : adatokat ír a képernyőre, számításokat végez, módosítja egy file tartalmát, nyomtat, stb.); végül a ***befejező szakasz*** : ez a műveletek megfelelő lezárására szolgál (vagyis a nyitva maradt file-ok lezárása, külső kapcsolatok bontása, stb.)

« Szövegmódú » programban ez a három szakasz a szemközti ábra lineáris szervezését követi. Ennek az a következménye, hogy ezeket a programokat nagyon korlátozott interaktivitás jellemzi. A felhasználónak gyakorlatilag semmilyen szabadsága sincs : a program időről-időre megkéri, hogy írjon be adatokat, de mindíg a programutasítások sorrendjének megfelelő, előre meghatározott sorrendben.

Egy grafikus interface-ű programnak ettől eltérő a belső szervezése. Azt mondjuk, hogy az ilyen program ***eseményvezérelt***. Az inicializálási szakasz után az ilyen típusú program « várakozásba megy át » és rábízza magát egy másik, többé-kevésbé mélyen a számítógép operációs rendszerébe integrált és állandóan működő programra.

Ez az ***eseményfogadó*** állandón figyeli a perifériákat (billentyűzetet, egeret, órát, modemet, stb.) és azonnal reagál rá, ha egy eseményt érzékel.
Egy ilyen esemény lehet bármilyen felhasználói akció: az egér elmozdítása, billentyű lenyomása, stb., de egy belső esemény, vagy egy automatizmus is lehet (például órajel).



Amikor az eseményfogadó érzékel egy eseményt, akkor egy specifikus jelet küld a programnak[30], aminek azt fel kell ismerni, hogy reagáljon rá.

Egy grafikus user interface-ű program inicializálási szakasza olyan utasításokból áll, amik az interface különböző interaktív komponenseit helyezik el (ablakokat, gombokat, check boxokat, stb.). Más utasítások a kezelendő üzeneteket definiálják : eldönthetjük, hogy a program csak bizonyos eseményekre reagáljon és a többit hagyja figyelmen kívül.

Míg egy « szöveges » programban a működési szakasz olyan utasítás sorozatból áll, ami előre leírja azt a sorrendet, ahogyan a gépnek a különböző feladatait végre kell hajtania (még ha különböző végrehajtási utakról gondoskodunk is válaszul a végrehajtás során adódó különböző feltételekre), addig egy grafikus interface-ű program működési fázisában csak független függvényeket találunk. Ezeket a program specifikusan akkor hívja, amikor egy meghatározott eseményt érzékel az operációs rendszer : vagyis azt a feladatot végzik el, amit erre az eseményre válaszul várunk a programtól és semmi mást[31].

Fontos, hogy megértsük : ez alatt az eseményfogadó folyamatosan működik és esetleges más események bekövetkezésére vár.

Ha más események következnek be, akkor megtörténhet, hogy egy második (vagy egy 3., egy 4.) függvény aktiválódik és az elsővel, ami még nem fejezte be működését[32], párhuzamosan kezd el működni. A modern operációs rendszerek és nyelvek lehetővé teszik ezt a párhuzamosságot, amit ***multitaskingnak*** is nevezünk.

Az előző fejezetben már említettem, hogy a programszöveg struktúrája nem adja meg közvetlenül az utasítások végrehajtási sorrendjét. Ez a megjegyzés méginkább igaz a grafikus interface-ű programokra, mivel a függvények hívásának sorrendje már sehol sincs leírva a programban. Az események vezérelnek !

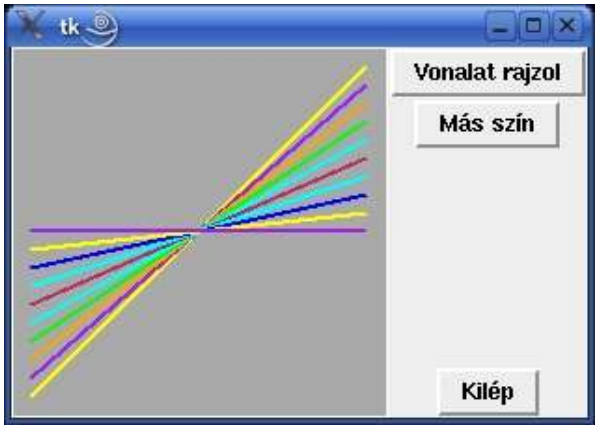
Ez biztosan bonyolultnak tűnik. Néhány példával fogom illusztrálni.

30 Ezeket az üzeneteket gyakran WM-mel jelöljük (Window messages: Window üzenetek) egy (reaktív zónákat : gombokat, checkboxokat, legördülő menüket, stb. tartalmazó) ablakokból álló grafikus környezetben. Az algoritmusok leírásakor gyakran előfordul, hogy összekeverjük ezeket az üzeneteket magukkal az eseményekkel.

31 Szígorúan véve az olyan függvényeket, amik nem küldenek vissza semmilyen értéket sem, ***eljárásnak*** (***procedurának***) nevezzük. (lásd 54. oldal lábjegyzetét).

32 Azonos események megjelenésére válaszul többször meghívható ugyanaz a függvény, vagyis ugyanazt a feladatot több konkurens példány hajtja végre. A későbbiekben majd meglátjuk, hogy ez nemkívánatos mellékhatásokat eredményezhet.

8.3.1 Grafikus példa: vonalak rajzolása vászonra



Az alábbi script egy ablakot hoz létre, ami három nyomógombot és egy ***vásznat*** tartalmaz. A *Tkinter* terminológiája szerint a vászon egy téglalap alakú felület, amibe speciális metódusokkal[33] különböző rajzokat és képeket lehet elhelyezni.

A « Vonalat rajzol » gombra kattintva a vásznon megjelenik egy új, színes vonal, ami minden alkalommal elhajlik az előzőtől.

Ha a « Más szín » gombra kattintunk, akkor egy véges színkészletből véletlenszerűen egy új színt választ a script. Ez lesz a következő egyenes színe.

A « Kilép » gomb az alkalmazás befejezésére és az ablak zárására való.

```
# Tkinter grafikus könyvtárat alkalmazó gyakorlat

from Tkinter import *
from random import randrange

# --- az eseménykezelő függvények definíciója : ---
def drawline():
    "Vonal rajzolása a can1 canvasra (vászonra)"
    global x1, y1, x2, y2, color
    can1.create_line(x1,y1,x2,y2,width=2,fill=color)

    # a koordináták módosítása a következő egyenes számára :
    y2, y1 = y2+10, y1-10

def changecolor():
    "a rajz színének véletlenszerű megváltoztatása"
    global color
    pal=['purple','cyan','maroon','green','red','blue','orange','yellow']
    c = randrange(8)         # => véletlenszám generálása 0 és 7 között
    color = pal[c]

#------ Főprogram -------

# a következő változókat globális változókként használjuk :
x1, y1, x2, y2 = 10, 190, 190, 10      # az egyenes koordinátái
color = 'dark green'                   # az egyenes színe

# A fő-widget létrehozása ("master") :

abl1 = Tk()
# a "slave" widget-ek létrehozása :
can1 = Canvas(abl1,bg='dark grey',height=200,width=200)
can1.pack(side=LEFT)
```

33 Ezek a rajzok esetleg egy későbbi fázisban animáltak lehetnek (lásd később)

```
gomb1 = Button(abl1,text='Kilép',command=abl1.quit)
gomb1.pack(side=BOTTOM)
gomb2 = Button(abl1,text='Vonalat rajzol',command=drawline)
gomb2.pack()
gomb3 = Button(abl1,text='Más szín',command=changecolor)
gomb3.pack()

abl1.mainloop()         # eseményfogadó indítása

abl1.destroy()          # az ablak (destruction) zárása
```

A program működését – az előző oldalak magyarázatának megfelelően – két függvény biztosítja: a **drawline()** és a **changecolor()**, amiket az inicializálási szakaszban definiált események aktiválják.

Az inicializálási szakasz a teljes *Tkinter* modul és a *random* modul egy függvényének - ami véletlen számokat állít elő - az importálásával kezdődik. Ezután a **Tk()**, **Canvas()** és **Button()** osztályokból widgeteket hozunk létre.

Az inicializálás az eseménykezelőt indító **abl1.mainloop()** utasítással fejeződik be. A Python a következő utasításokat csak akkor hajtja végre, amikor kilép ebből a hurokból. A kilépést az **abl1.quit()** metódus indítja el (lásd alább)

A gombokat létrehozó utasításban a **command** opcióval adhatjuk meg azt a függvényt, amit egy < *bal egérgomb kattintás a widgetre* > esemény bekövetkezésekor kell hívni. A *Tkinter,* hogy megkönnyítse a dolgunkat, ezt a megoldást kínálja, mivel ezt az eseményt kapcsoljuk természetes módon egy gomb típusú widgethez. A későbbiekben meg fogjuk látni, hogy vannak más, általánosabb technikák, amikkel bármilyen eseményt hozzákapcsolhatunk bármelyik *widget*-hez.

A script függvényei a főprogramban definiált változók értékeit megváltoztathatják. Ezt a függvénydefiníciókban használt **global** utasítás teszi lehetővé. Egy darabig így járunk el (de csak azért, hogy hozzászokjunk a lokális és globális változók viselkedése közötti különbséghez), azonban a későbbiekben meg fogjuk érteni : ennek a technikának az alkalmazása egyáltalán nem javasolt, különösen akkor nem, ha nagyméretű programokat írunk. Amikor rátérünk az osztályok tanulmányozására, egy jobb technikát fogunk megtanulni (a 154. oldaltól).

A « Kilép » gombhoz kapcsolt parancs az **abl1** ablak **quit()** metódusát hívja. Ez az ablakhoz kapcsolt eseményfogadóból (*mainloop*) való kilépésre szolgál. Aktiválásakor a programvégrehajtás a *mainloop* -ot követő utasításokkal folytatódik. Példánkban ez az ablak törléséből (*destroy*) áll.

(8) Gyakorlatok : az előző « Vonalrajzoló » program módosításai.

8.1. Hogyan kell módosítani a programot, hogy többé ne legyen cian, maroon és green (cián, barna és zöld) vonal ?

8.2. Hogyan kell módosítani a programot, hogy minden vonal vízszintes és párhuzamos legyen ?

8.3. Nagyítsa fel a vásznat. Legyen a szélessége 500, a hosszúsága 650 egység. Változtassa meg a vonalak hosszát is úgy, hogy a vászon széléig érjenek.

8.4. Adjon a programhoz egy « drawline2 » függvényt, ami egy vörös keresztet rajzol a vászon közepére. Adjon egy « Kereső » nevű gombot is a programhoz. A gombra való kattintással kell kirajzolni a keresztet !

8.5. Az eredeti programban helyettesítsük a « drawline » (« rajzolj egyenest ») metódust a « drawrectangle » (« rajzolj téglalapot ») metódussal. Mi történik ?
Ugyanígy próbálja ki a « drawarc » (« rajzolj körívet ») « drawoval » (« rajzolj ellipszist ») « drawpolygon »(« rajzolj sokszöget ») metódust.
Mindegyik módszernél adja meg, hogy mit jelölnek a zárójelben megadott paraméterek !

(Megjegyzés : a sokszög esetében a programon módosítani kell !)

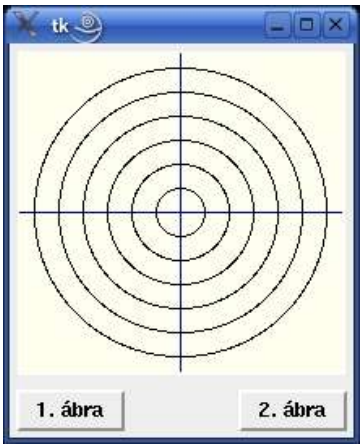
8.6. - Törölje a « global x1, y1, x2, y2 » sort az eredeti program « drawline » függvényében. Mi történik ? Miért ?
- Ha a « drawline » függvényt definiáló sorban az « x1, y1, x2, y2 » -t zárójelbe teszi, függvényparaméterként átadva ezeket a változókat, fog-e még működni a program ? (Ne felejtsük el módosítani a függvényhívó programsort sem !)
- Mi történik, ha « x1, y1, x2, y2 = 10, 390, 390, 10 » értékadó utasítás van a « global x1, y1, ... »utasítás helyén ? Miért ? Milyen következtetést tud ebből levonni ?

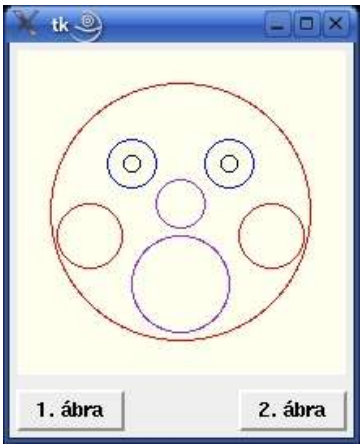
8.7. a) Írjon egy rövid programot, ami egy fehér hátterű (white) téglalapba rajzolja az 5 olimpiai karikát. Egy « Kilépés » gombnak kell bezárni az ablakot.
b) Módosítsa a fenti programot úgy, hogy hozzátesz 5 gombot. Minden egyes gomb rajzoljon egy karikát.

8.8. Készítsen a füzetében egy kétoszlopos táblázatot. A baloldali oszlopba írja be a már megismert osztályok definícióit (a paraméterlistájukkal együtt) és a jobboldali oszlopba az osztályok metódusait (a paraméterlistájukkal együtt). Hagyjon helyet a későbbi kiegészítésnek.

8.3.2 Grafikus példa : rajzok közötti váltás

Ez a példa azt mutatja be, hogyan lehet felhasználni a programhurkokról, listákról és függvényekről szerezett ismereteket arra, hogy néhány kódsor segítségével több rajzot hozzunk létre. Az alkalmazásunk a választott gombtól függően fogja megjeleníteni az alábbi rajzok közül az egyiket.






```
from Tkinter import *

def circle(x, y, r, color ='black'):
    "r sugarú (x,y) középpontú lör rajzolása"
    can.create_oval(x-r, y-r, x+r, y+r, outline=color)

def figure_1():
    "celtablat rajzol"
    # először a meglévő rajz törlése :
    can.delete(ALL)
    # a két egyenes rajzolása (függ. és vizsz.) :
    can.create_line(100, 0, 100, 200, fill ='blue')
    can.create_line(0, 100, 200, 100, fill ='blue')
    # több koncentrikus kör rajzolása :
    radius = 15
    while radius < 100:
        circle(100, 100, radius)
        radius += 15


def figure_2():
    "egyszerűsített arc rajzolása"
    # először minden meglévő rajz törlése :
    can.delete(ALL)
    # minden kör jellemzőjét listák listájába
    # tesszük :
    cc =[[100, 100, 80, 'red'],     # fej
         [70, 70, 15, 'blue'],      # szem
         [130, 70, 15, 'blue'],
         [70, 70, 5, 'black'],
         [130, 70, 5, 'black'],
         [44, 115, 20, 'red'],      # arc
         [156, 115, 20, 'red'],
         [100, 95, 15, 'purple'],   # orr
         [100, 145, 30, 'purple']]  # száj
    # az osszes kort egy ciklus segitsegevel rajzoljuk meg :
    i =0
    while i < len(cc):      # a lista bejárása
        el = cc[i]          # minden elem maga is lista
        circle(el[0], el[1], el[2], el[3])
        i += 1

##### Főprogramm : ############

window = Tk()
can = Canvas(window, width =200, height =200, bg ='ivory')
can.pack(side =TOP, padx =5, pady =5)
b1 = Button(window, text ='1. ábra', command =figure_1)
b1.pack(side =LEFT, padx =3, pady =3)
b2 = Button(window, text ='2. ábra', command =figure_2)
b2.pack(side =RIGHT, padx =3, pady =3)
window.mainloop()
```

Kezdjük a script végén lévő főprogram elemzésével : Létrehozunk a **Tk()** osztályból egy ablak-objektumot a **window** változóban.
Utána létrehozunk ebben az ablakban 3 widgetet : egy vásznat (canvas) a **can** és két gombot (button) a **b1** és **b2** változókba. A *widget* -eket - mint az előző scriptben - a **pack()** metódusukkal pozícionáljuk az ablakban, de most a metódust opciókkal használjuk :

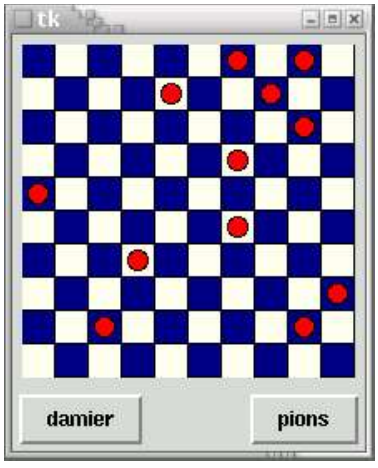
- a **side** opció a TOP, BOTTOM, LEFT és RIGHT értékeket fogadja el. A widget-et a megfelelő oldalra teszi az ablakban.
- a **padx** és **pady** opciók egy sávot tartanak szabadon a *widget* körül. Ezt a sávot pixelekben adjuk meg : a **padx** a *widget* bal- és jobboldalán, a **pady** a *widget* alatt és fölött foglalja le az említett sávot.

A gombok vezérlik a két ábra kirajzolását a **figure_1()** és **figure_2()** függvények hívásával. Mivel több kört kell rajzolni a vászonra, ezért úgy gondolom, hasznos lenne először egy specializált **circle()** függvényt írni. Valószínűleg az olvasó már tudja, hogy a *Tkinter* canvas-nak van egy **create_oval()** metódusa, amivel akármilyen ellipszist (így köröket is) rajzolhatunk. Azonban ezt a metódust négy argumentummal kell hívni. Az argumentumok egy fiktív téglalap balfelső és jobbalsó sarkának a koordinátáit adják meg, amibe bele fogjuk rajzolni az ellipszist. A kör speciális esetében ez nem túl praktikus. Természetesebbnek tűnik a rajzolást a kör középpontjának és sugarának megadásával vezérelni. Ezt a **circle()** függvényünkkel érjük el, ami a koordináták konverzióját elvégezve hívja a **create_oval()** metódust. Vegyük észre, hogy a függvény egy opcionális argumentumot vár, ami a rajzolandó kör színére vonatkozik (ez alapértelmezetten fekete).

A **figure_1()** függvényben világosan szembetűnik ennek a megközelítésnek a hatékonysága. Egy programhurok van benne, ami a(z azonos középpontú, növekvő sugarú) körök sorozatának kirajzolására szolgál. Figyeljük meg a += incrementáló operátor használatát (példánkban az r változó értékét növeli 15 egységgel minden iterrációban).

A második rajz egy kicsit összetettebb, mert változó méretű, különböző középpontú körökből van összeállítva. Az összes kört ugyanígy, egyetlen programhurok segítségével rajzolhatjuk meg, ha felhasználjuk a listákról szerzett tudásunkat.

Négy jellemző különbözteti meg egymástól a megrajzolandó köröket :a középpont x és y koordinátái, a sugár és a szín. Mindegyik körnek ezt a négy jellemzőjét összegyűjthetjük egy-egy listában és ezeket a listákat egy másik listában tárolhatjuk. Így egy ***egymásba ágyazott listánk*** lesz, amit elég lesz egy ciklus segítségével bejárnunk a megfelelő rajzok elkészítéséhez.

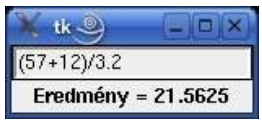


Gyakorlatok :

8.9. Irjon egy scriptet az előző alapján, ami egy sakktáblát (fehér alapon fekete négyzeteket) jelenít meg, amikor egy gombra kattintunk :

8.10. Az előző gyakorlatba építsünk be még egy gombot, ami korongokat jelenít meg véletlenszerűen a sakktáblán (a gomb minden megnyomására egy új korong jelenjen meg).

8.3.3 Grafikus példa: egy egyszerű számológép



Bár a következő script nagyon rövid, mégis egy komplett számológépet implementál, amivel zárójeleket és tudományos függvényeket tartalmazó számításokat végezhetünk. Semmi rendkívülit nem látunk. Ezek a funkciók csak annak a ténynek a következményei, hogy a program végrehajtására interpretert, nem pedig fordítóprogramot használunk.

Tudjuk, hogy a fordítóprogram csak egyszer működik azért, hogy végrehajtható kódra fordítsa le a forráskódot. A szerepe tehát még a programvégrehajtás ***előtt*** befejeződik. Az interpreter viszont aktív a programvégrehajtás ***alatt*** és rendelkezésre áll, hogy bármilyen új forráskódot - mint például a felhasználó által beírt matematikai kifejezést - lefordítson.

Az interpretált nyelveknek mindíg vannak olyan függvényei, amik lehetővé teszik egy stringnek a nyelv utasítássoraként történő kiértékelését. Így dinamikus programstruktúrákat hozhatunk létre néhány sorban. Az alábbi példában az **eval()** beépített függvényt használjuk a felhasználó által – az erre a célra szánt mezőbe – beírt matematikai kifejezés elemzésére és utána nincs más dolgunk, mint kiíratni az eredményt.

```
# A Tkinter grafikus könyvtárat és a math modult alkalmazó gyakorlat

from Tkinter import *
from math import *

# annak az akciónak a definíciója, amit akkor kell végrehajtani, ha a
# felhasználó az adatbeviteli mező kitöltése után «Enter»-t nyom

def kiertekel(event):
    chain.configure(text = "Eredmény = " + str(eval( mezo.get())))

# ----- Főprogram : -----

ablak = Tk()
mezo = Entry( ablak)
mezo.bind("<Return>", kiertekel)
mezo.pack()
chain = Label( ablak)
chain.pack()

ablak.mainloop()
```

A script elején a *Tkinter* és a *math* modulokat importáljuk. Az utóbbira azért van szükség, hogy az említett számológépnek valamennyi használatos matematikai és tudományos függvény (sin, cos, négyzetgyök, stb.) rendelkezésére álljon.

Ez után definiáljuk a **kiertekel()** függvényt. Ezt a program akkor fogja végrehajtani, amikor a felhasználó a *Return* (vagy *Enter*) gombra kattint, miután a - későbbiekben leírt - adatbeviteli mezőbe beírt valamilyen matematikai kifejezést.

Ez a függvény a **chain**[34] widget **configure()** metódusát használja a **text** attribútum módosítására. Az attribútum új értékét az határozza meg, amit az egyenlőségjel jobboldalára írtunk. A Pythonba beépített két függvény, az **eval()** és a **str()** valamint egy *Tkinter* widget-hez kapcsolódó **get()** metódus segítségével dinamikusan konstruált stringről van szó.

Az **eval()** az interpretert hívja az argumentumként egy karakterláncban átadott Python-kifejezés kiértékeléséhez. A kiértékelés eredménye az **eval()** visszatérési értéke. Példa :

```
chain = "(25 + 8)/3"      # chaîn egy matematikai kifejezést tartalmaz
res = eval(chain)         # chaîn-ben levő kifejezés kiértékelése
print res +5              # => a res változó tartalma numerikus
```

A **str()** egy numerikus kifejezést alakít karakterlánccá. Azért kell ezt a függvényt hívni, mert az előző függvény (az eval()) egy numerikus értéket ad vissza, amit újra karakterlánccá alakítunk, hogy a « Eredmény = » üzenetbe bele tudjuk foglalni.

A **get()** metódus az **Entry** class *widget* -jeihez kapcsolódik. Programunkban egy ilyen típusú *widgettel* tesszük lehetővé a felhasználónak, hogy bármilyen numerikus kifejezést beírhasson a billentyűzet segítségével. A **get()** metódussal lehet « kiszedni » a « **mezo** » *widget* -ből a felhasználó által megadott karakterláncot.

A főprogram tartalmazza az inicializálási szakaszt, ami az eseményfigyelő (*mainloop*) indításával végződik. A főprogramban létrehozunk egy **Tk()** ablakot, ami a **Label()** class-ból létrehozott « **chain** » és az **Entry()** class-ból létrehozott « **mezo** » *widget* -et tartalmazza.

Figyelem : azért, hogy a « **mezo** » *widget* valóban tenni tudja a dolgát, vagyis hogy a programnak át tudja adni azt a kifejezést, amit a felhasználó beírt, a **bind()**[35] metódus segítségével ***hozzá kapcsolunk egy eseményt*** :

```
mezo.bind("<Return>",kiertekel)
```

Ez az utasítás a következőt jelenti : « *Kösd a <Return> billentyű lenyomásának eseményét a <mezo> objektumhoz és kezelje ezt az eseményt a <kiertekel> függvény* »

A kezelendő eseményt egy speciális karakterlánc írja le (példánkban a « <Return> » string). Számos esemény létezik (egérmozgások és kattintások, billentyűlenyomás, ablakok elhelyezése és átméretezése, stb.). Minden esemény speciális stringje megtalálható a *Tkinter* -rel foglalkozó referencia művekben.

Használjuk ki az alkalmat arra, hogy mégegyszer megfigyeljük egy objektum metódushívásának a szintaxisát :

objektum.metódus(argumentumok)

Először leírjuk az objektum nevét, azt követi egy pont (ami operátorként működik), majd a hívott metódus neve, végül zárójelben az argumentumok.

34 A **configure()** metódus bármely már létező widgetre alkalmazható a tulajdonságok módosítására.

35 A bind szó jelentése : összeköt

8.3.4 Grafikus példa : egérkattintás detektálása és helyének azonosítása

Az előző példa « kiertekel » függvényének definíciójánál megjegyeztem, hogy az **event** (esemény) argumentumot adtuk meg zárójelben.

Ez az argumentum kötelező. Amikor egy eseménykezelő függvényt definiálunk, amit valamelyik widget-hez kötünk annak **bind()** metódusával, akkor mindíg meg kell adni argumentumként az eseményt. Egy automatikusan létrehozott standard Python-objektumról van szó, ami lehetővé teszi, hogy az eseménykezelőnek átadjuk ennek az eseménynek az attribútumait :

- az eseménytípust : az egér elmozdulása, az egérgombok lenyomása vagy elengedése, egy billentyű lenyomása a klaviatúrán, a kurzor egy előre definiált zónába kerülése, egy ablak megnyitása vagy zárása, stb.
- az esemény tulajdonságait : az esemény keletkezésének pillanata, koordinátái, az érintett widget (ek) jellemzői, stb.

Nem fogunk nagyon belemerülni a részletekbe. Ha az olvasó beírja az alábbi scriptet és kísérletezik vele, akkor gyorsan meg fogja érteni az elvet.

```
# Egérkattintás észlelése és helyének meghatározása egy ablakban :

from Tkinter import *

def mutato(event):
    chain.configure(text = "Kattintás detektálva: X =" + str(event.x) +\
                           ", Y =" + str(event.y))

ablak = Tk()
keret = Frame(ablak, width =200, height =150, bg="light yellow")
keret.bind("<Button-1>", mutato)
keret.pack()
chain = Label(ablak)
chain.pack()

ablak.mainloop()
```

A script egy halványsárga **keretet** (*frame*) tartalmazó ablakot jelenít meg, amibe a felhasználónak az egérrel bele kell kattintani.

A <kattintás az egér első gombjával> eseményt a frame-widget **bind()** metódusa kapcsolja a « mutato » eseménykezelőhöz.

Ez az eseménykezelő a Python által automatikusan létrehozott **event** objektum **x** és **y** attribútumait használhatja annak a stringnek a megkonstruálására, ami az egér pozícióját fogja tartalmazni a kattintás pillanatában.

Gyakorlat :

8.11. Módosítsa úgy a fenti scriptet, hogy az egy piros kört jelenítsen meg a kattintás helyén (először a Frame widget-et egy Canvas widget-tel kell helyettesíteni).

8.4 A Tkinter widget-osztályai

***Megjegyzés** : A tanfolyam a során apránként bemutatom néhány widget használatát. Azonban nem áll szándékomban egy komplett Tkinter kézikönyvet írni. Azokra a widget-ekre fogok szorítkozni, amik módszertani szempontból a legérdekesebbnek tűnnek, vagyis amik segíthetnek megérteni olyan fontos fogalmat, mint amilyen az osztály (class) fogalma. Aki többet akar tudni, az szíveskedjen az irodalomban (lásd 8. oldal) utána nézni.*

A *Tkinter* widget-eknek 15 alaposztálya létezik :

Widget	*Leírás*
Button	Klasszikus nyomógomb, valamilyen utasítás végrehajtásának az előidézésére használják.
Canvas	Különböző grafikus elemek elhelyezésére szolgáló felület. Rajzolásra, grafikus szerkesztők létrehozására és testre szabott widget-ek implementálására is használhatjuk.
Checkbutton	Egy jelölőnégyzet, aminek két különböző állapota lehet (a négyzet ki van jelölve vagy nincs kijelölve). Egy klikkelés a widgeten állapotváltozást idéz elő.
Entry	Adatbeviteli mező, amibe a felhasználó bármilyen szöveget beírhat.
Frame	Egy téglalap alakú felület az ablakban, ahova más widget-eket tehetünk. Ez a felület színes lehet. Szegélye is lehet.
Label	Valamilyen szöveg (vagy címke) (esetleg egy kép).
Listbox	A felhasználónak - általában valamilyen doboz formájában - felajánlott választéklista. A Listbox-ot úgy is konfigurálhatjuk, hogy vagy egy rádiógomb vagy egy jelölőnégyzet sorozatot tartalmazzon.
Menu	Menü. Lehet címsorhoz kapcsolódó legördülő menü, vagy egy kattintás után akárhol feljövő úszó « *pop up* » menü.
Menubutton	Menügomb, amit legördülő menük implementálására használnak.
Message	Szöveg kiírását teszi lehetővé. A Label widget egy változata, ami lehetővé teszi, hogy a kiírt szöveg automatikusan egy bizonyos mérethez, vagy szélesség/magasság arányhoz igazodjon.
Radiobutton	(Egy fekete pont egy kis körben.) Egy változó lehetséges értékeit reprezentálja. Az egyik rádiógombra való kattintás az annak megfelelő értéket adja a változónak.
Scale	Egy kurzornak egy skála mentén való mozgatásával teszi láthatóvá egy változó értékének a változtatását.
Scrollbar	A görgető sort más widget-ekhez (Canvas, Entry, Listbox, Text) kapcsolva használhatjuk.
Text	Formázott szöveg kiírása. A felhasználónak is lehetővé teszi a kiírt szöveg formázását. Képeket is be lehet szúrni.
Toplevel	Egy külön, felülre kiírt ablak.

Mindegyik *widget*-osztály-nak számos beépített metódusa van. Kapcsolhatunk hozzájuk eseményeket is, amint azt az előző oldalakon láttuk. Többek között meg fogjuk tanulni, hogy ezeket a widget-eket a **grid()**, a **pack()** és a **place()** metódusok segítségével pozícionálhatjuk az ablakokban.

Ezeknek a módszereknek a haszna akkor válik nyilvánvalóvá, amikor megpróbálunk ***portábilis*** programokat írni (vagyis olyanokat, amik képesek különböző operációs rendszereken – mint a *Unix*, *MacOS* vagy *Windows* - futni), amiknek az ablakai átméretezhetőek.

8.5 A grid() metódus alkalmazása widget-ek pozícionálására

Eddig mindíg a **pack()** metódus segítségével helyeztük el a widget-eket az ablakukban. Ennek a metódusnak a rendkívüli egyszerűség az előnye, azonban nem ad túl sok szabadságot a widget-ek kedvünk szerinti elrendezéséhez. Mit kell például csinálni, hogy megkapjuk az ábrán látható elrendezést ?

Tehetnénk néhány kísérletet, mint amilyeneket az előzőekben tettünk, hogy a **pack()** metódus nak « side = » típusú argumentumokat adunk, de ez nem vezetne messzire. Próbáljuk meg például :

```
from Tkinter import *

abl1 = Tk()
txt1 = Label(abl1, text = 'Első mező :')
txt2 = Label(abl1, text = 'Második :')
mezo1 = Entry(abl1)
mezo2 = Entry(abl1)
txt1.pack(side =LEFT)
txt2.pack(side =LEFT)
mezo1.pack(side =RIGHT)
mezo2.pack(side =RIGHT)

abl1.mainloop()
```

...az eredmény nem igazán az, amit kerestünk !!! :

Hogy jobban megértsük a **pack()** metódus működését, megpróbálhatjuk még a **side =TOP, side=BOTTOM** opciók különböző kombinációit a négy widget mindegyikére. Azonban biztosan nem fogjuk megkapni a kívánt elrendezést. Két kiegészítő **Frame()** widget definiálásával és a **Label ()** és **Entry()** widget-ek ezekben történő elhelyezésével talán sikerülne eljutni a kívánt elrendezéshez, azonban ez nagyon bonyolult lenne.

Itt az ideje, hogy megtanuljuk a probléma egy másik megközelítését. Elemezzük az alábbi scriptet: (ez már majdnem a megoldás) :

```
from Tkinter import *

abl1 = Tk()
txt1 = Label(abl1, text = 'Első mező :')
txt2 = Label(abl1, text = 'Második :')
mezo1 = Entry(abl1)
mezo2 = Entry(abl1)
txt1.grid(row =0)
txt2.grid(row =1)
mezo1.grid(row =0, column =1)
mezo2.grid(row =1, column =1)
abl1.mainloop()
```

Ebben a scriptben a **pack()** metódust a **grid()**-del helyettesítettük. Látható, hogy az utóbbi használata igen egyszerű. Ez az ablakot egy táblázatnak (rácsnak) tekinti. Elég, ha megadjuk neki, hogy a táblázat melyik sorába (*row*) és melyik oszlopába (*column*) akarjuk elhelyezni a *widget*-eket. A sorokat és oszlopokat úgy számozhatjuk, ahogy akarjuk : kezdhetjük nullától, vagy egytől, vagy akármilyen számtól. A *Tkinter* figyelmen kívül fogja hagyni az üres sorokat és oszlopokat. Jegyezzük meg, ha nem adunk meg semmilyen sor vagy oszlopszámot, akkor az alapértelmezett érték nulla lesz.

A Tkinter automatikusan meghatározza a szükséges sorok és oszlopok számát. De ez nem minden : ha figyelmesen megvizsgáljuk a fenti scripttel létrehozott ablakot, látni fogjuk, hogy még nem értük el a kitűzött célunkat. Az ablak baloldalán megjelenő karakterláncok centráltak, míg mi jobbra igazítva szeretnénk őket. Ahhoz, hogy ezt elérjük, elég a **grid()** metódusnak egy argumentumot megadni. A **sticky** opció négy értéket vehet fel: **N, S, W, E** (a négy égtáj angol rövidítését). Ettől az értéktől függően lesznek a *widget*-ek föntre, lentre, balra, vagy jobbra igazítva. Helyettesítsük a scriptben a két első **grid()** utasítást a következőkkel :

```
txt1.grid(row =0, sticky =E)
txt2.grid(row =1, sticky =E)
```

... és pontosan a kívánt elrendezést fogjuk megkapni.

Elemezzük most a következő ablakot :

Az ablak 3 oszlopot tartalmaz: az elsőben 3 string, a másodikban 3 adatbeviteli mező, a harmadikban egy kép van. Az első két oszlopoban 3-3 sor van, viszont a harmadik oszlopban lévő kép valamilyen módon lefedi a három sort.

A következő a kódja :

```
from Tkinter import *

abl1 = Tk()

# a 'Label' és 'Entry' widgetek létrehozása:
txt1 = Label(abl1, text ='Első mező :')
txt2 = Label(abl1, text ='Második :')
txt3 = Label(abl1, text ='Harmadik :')
mezo1 = Entry(abl1)
mezo2 = Entry(abl1)
mezo3 = Entry(abl1)

# egy bitmap képet tartalmazó 'Canvas' widget létrehozása :
can1 = Canvas(abl1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_p.gif')
item = can1.create_image(80, 80, image =photo)

# laptördelés a'grid' metódus segítségével :
txt1.grid(row =1, sticky =E)
txt2.grid(row =2, sticky =E)
txt3.grid(row =3, sticky =E)
mezo1.grid(row =1, column =2)
mezo2.grid(row =2, column =2)
mezo3.grid(row =3, column =2)
can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)

# indítás :
abl1.mainloop()
```

Ahhoz, hogy az olvasó futtatni tudja a scriptet, valószínűleg a képfile nevét (Martin_p.gif) a választása szerinti kép nevével kell helyettesítenie. Figyelem : a standard *Tkinter* könyvtár csak kevés fileformátumot fogad el. Válassza lehetőség szerint a GIF formátumot.

Néhány dolgot megjegyezhetünk a scriptből :

1. A kép beszúrására alkalmazott technikát :
   A *Tkinter* nem teszi lehetővé, hogy közvetlenül szúrjunk be képet az ablakba. Először egy vásznat (canvas) kell az ablakba elhelyezni és a képet erre kell tenni. A vászonnak fehér színt választottam azért, hogy meg lehessen különböztetni az ablaktól. Ha azt akarjuk, hogy a vászon láthatatlanná váljon, akkor a **bg ='white'** paraméteret **bg ='gray'**-vel helyettesítsük. Mivel többféle képtípus létezik, ezért a **PhotoImage()**[36] class-ban a képobjektumot GIF típusú bitmap képnek kell deklarálnunk.

2. A kép vászonra helyezésének módját :
   ```
   item = can1.create_image(80, 80, image =photo)
   ```
   Pontos szóhasználattal azt mondjuk, hogy a **can1** ***objektumhoz*** (amely objektum a **Canvas** osztály egy ***példánya***) kötött **create_image()** ***metódust*** használjuk. A két első ***argumentum*** (**80, 80**) adja meg a vászonnak azt az x és y koordinátáját, ahová a kép középpontját kell tenni. (Mivel a vászon 160x160-as méretű, ezért a képet a vászon közepére centrálja a koordináta választásunk).

3. A sorok és oszlopok számozását a **grid()** metódusban :
   Megállapíthatjuk, hogy ez alkalommal a **grid()** metódusban a sorok és oszlopok számozása 1-gyel kezdődik (nem pedig nullával, mint az előző scriptben). Amint már feljebb említettem, a számozás választása teljesen szabad.
   Számozhatnánk így is : 5, 10, 15, 20... mert a *Tkinter* figyelmen kívül hagyja az üres sorokat és oszlopokat. Az 1-től kezdődő számozás valószínűleg a kódunk olvashatóságát javítja.

4. A **grid()**-del a vászon elhelyezésére használt argumentumokat :
   ```
   can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)
   ```
   Az első kettő azt jelenti, hogy a vászon a harmadik oszlop első sorában lesz elhelyezve. A harmadik (**rowspan =3**) azt jelenti, hogy három sorra fog kiterjedni..
   A két utolsó (**padx =10**, **pady =5**) annak a felületnek a méreteit jelenti, amit a widget körül kell lefoglalnunk (magasságban és szélességben).

5. És ha már itt tartunk, használjuk ki ezt a jól kivesézett példát arra, hogy megtanuljuk egy kicsit egyszerűsíteni a kódunkat ...

36 A PhotoImage osztály standard implementációja GIF és PGM/PPM képeket tud beolvasni. (*ford. megjegyzése*) Léteznek más kép class-ok, de azok használatához más grafikus modulokat kell importálni a scriptbe. Például a PIL (*Python Imaging Library*) -lel lehet kísérletezni.

8.6 Utasítások komponálása a tömörebb kód érdekében

Abból adódóan, hogy a Python magasszintű programozási nyelv, gyakran lehetséges (és kívánatos) azért átdolgozni egy scriptet, hogy tömörebb legyen.

Például gyakran használhatjuk az utasítások kombinálását arra, hogy a widget-ekre létrehozásásuk pillanatában a (**grid()**, **pack()** vagy **place**)) pozícionáló metódusokat alkalmazzuk. A kód egyszerűbbé és néha olvashatóbbá válik. Például az előző script két sorát :

```
txt1 = Label(abl1, text ='Első mező :')
txt1.grid(row =1, sticky =E)
```

egyetlen sorral így helyettesíthetjük :

```
Label(abl1, text ='Első mező :').grid(row =1, sticky =E)
```

Ezzel az új írásmóddal megtakarítjuk a **txt1** közbenső változót. Ezt a változót arra használtuk, hogy megkönnyítettük eljárásunk egymást követő lépéseit, de nincs rá feltétlenül szükség. A **Label ()** osztály hívására még akkor is létrejön ennek az osztálynak egy objektuma, ha az objektum hivatkozását nem tároljuk egy változóban. (A *Tkinter* minden esetre megőrzi ezt a hivatkozást az ablak belső reprezentációjában.) Ha így járunk el, a hivatkozás elvész a script hátralévő része számára, de egyetlen egy összetett utasításban az objektum létrehozásának pillanatában mégis alkalmazható rá egy olyan pozícionáló metódus, mint a **grid()**. Nézzük meg ezt egy kicsit részletesebben :

Eddig úgy hoztunk létre különböző objektumokat valamilyen osztályból, hogy azokat minden alkalommal változókhoz rendeltük hozzá. Például amikor azt írtuk, hogy :

```
txt1 = Label(abl1, text ='Első mező :')
```

akkor a **Label()** osztály egy példányát hoztuk létre, amit a `txt1` változóhoz rendeltük hozzá.

A **txt1** változóval erre a példányra hivatkozhatunk a scriptben mindenhol. Valójában azonban csak egyszer használjuk, mégpedig arra, hogy alkalmazzuk rá a **grid()** metódust, mert a szóban forgó widget nem egyéb mint egy egyszerű címke. Nem javasolható gyakorlat csak azért létrehozni egy új változót, hogy utána csak egyszer (és közvetlenül a létrehozása után) hivatkozzunk rá, ugyanis a változó feleslegesen foglal le memóriaterületet.

Ha ilyen helyzet áll elő, akkor bölcsebb dolog összetett utasításokat alkalmazni. Például a következő két utasítást:

```
osszeg = 45 + 72
print osszeg
```

inkább egyetlen összetett utasítással helyettesítjük :

```
print 45 + 72
```

Így megtakarítunk egy változót.

Ugyanígy, amikor olyan widget-eket helyezünk el, amikre később nem kívánunk visszatérni, ami gyakran előfordul a **Label()** osztály widget-jei esetében, akkor azok létrehozásakor általában egyetlen összetett utasításban közvetlenül alkalmazhatók a pozícionáló metódusok (a **grid()** , **pack()** vagy **place()**).

Ez csak azokra a widget-ekre vonatkozik, amikre a létrehozásuk után többet nem hivatkozunk. ***Az összes többit változókhoz kell rendelni***, hogy a scriptben másutt még használni tudjuk őket.

Ez esetben két külön utasítást kell alkalmazni. Egyiket a widget létrehozására, a másikat a pozícionáló metódus alkalmazására. Nem képezhetünk olyan összetett utasítást, mint az alábbi :

```
mezo = Entry(abl1).pack()          # programozási hiba !!!
```

Ennek az utasításnak látszólag egy új widget-et kellett volna létrehozni, azt hozzárendelni a **mezo** változóhoz, a **pack()** metódus segítségével pozícionálni egyazon műveleten belül.
A valóságban az utasítás létrehozta az **Entry()** osztályból az új widget-et és a **pack()** metódus elvégezte az ablakbeli pozícionálást, de a **mezo** változóba a **pack()** metódus visszatérési értékét tárolta : ***ez pedig nem a widget hivatkozása***. Ezzel a visszatérési értékkel semmit sem tudunk kezdeni, ez egy üres objektum (None).

Hogy widget-hivatkozást kapjunk, két utasítást kell használnunk :

```
mezo = Entry(abl1)                 # a widget létrehozása
mezo.pack()                        # a widget pozícionálása
```

Megjegyzés : Ha a **grid()** metódust alkalmazzuk, akkor a sor- és oszlopszámok elhagyásával tovább egyszerűsíthetjük a programunkat. Attól a pillanattól kezdve, hogy a widget-ek pozícionálására a **grid()** metódust használjuk, a *Tkinter* úgy tekinti, hogy sorok és oszlopok[37] vannak. Ha nem adunk meg egy sor- vagy egy oszlopszámot, akkor a **grid()** metódus a megfelelő widget-et az első ***üres*** cellába teszi.

Az alábbi script tartalmazza az említett egyszerűsítéseket :

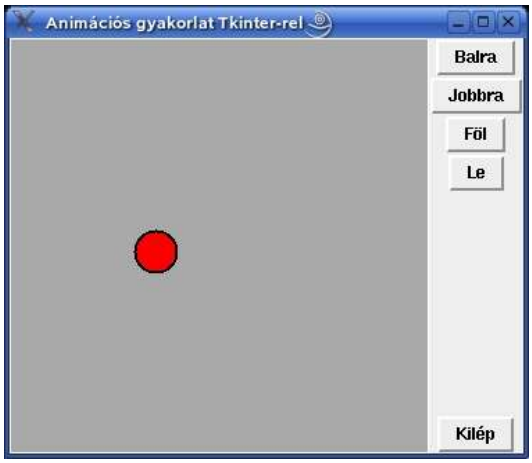
```
from Tkinter import *
abl1 = Tk()

# Label(), Entry(), és Checkbutton() widget-ek létrehozása :
Label(abl1, text = 'Első mező :').grid(sticky =E)
Label(abl1, text = 'Második :').grid(sticky =E)
Label(abl1, text = 'Harmadik :').grid(sticky =E)
mezo1 = Entry(abl1)
mezo2 = Entry(abl1)                  # ezekre a widget-ekre később
mezo3 = Entry(abl1)                  # hivatkozni fogunk :
mezo1.grid(row =0, column =1)        # ezért mindegyiket külön
mezo2.grid(row =1, column =1)        # változóhoz rendeljük
mezo3.grid(row =2, column =1)
chek1 = Checkbutton(abl1, text ='Checkbox a megjelenítéshez')
chek1.grid(columnspan =2)

# egy bitmap képet tartalmazó 'Canvas' (vászon) widget létrehozása:
can1 = Canvas(abl1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_p.gif')
can1.create_image(80,80, image =photo)
can1.grid(row =0, column =2, rowspan =4, padx =10, pady =5)

# indítás :
abl1.mainloop()
```

37 Ugyanabban az ablakban ne használjunk többféle pozícionáló metódust !
A grid(), pack() és place() metódusok kölcsönösen kizárják egymást.

8.7 Objektum tulajdonságainak módosítása - Animáció

Ezen a tudásszinten az olvasó bizonyára szeretne olyan programot írni, ami gombok segítségével egy rajzot mozgat egy vásznon.

Írjuk be, teszteljük és elemezzük a következő scriptet :




```
from Tkinter import *

# általános mozgatóeljárás :
def mozog(gd, hb):
    global x1, y1
    x1, y1 = x1 +gd, y1 +hb
    can1.coords(oval1, x1, y1, x1+30, y1+30)

# eseménykezelők :
def mozdit_balra():
    mozog(-10, 0)

def mozdit_jobbra():
    mozog(10, 0)

def mozdit_fel():
    mozog(0, -10)

def mozdit_le():
    mozog(0, 10)

#------ Főprogram -------

# a következő változók globálisak :
x1, y1 = 10, 10         # kiindulási koordináták

# A fő ("master") widget létrehozása :
abl1 = Tk()
abl1.title("Animációs gyakorlat Tkinter-rel")

# "slave" widget-ek létrehozása :
can1 = Canvas(abl1,bg='dark grey',height=300,width=300)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
can1.pack(side=LEFT)
Button(abl1,text='Kilép',command=abl1.quit).pack(side=BOTTOM)
Button(abl1,text='Balra',command=mozdit_balra).pack()
Button(abl1,text='Jobbra',command=mozdit_jobbra).pack()
Button(abl1,text='Föl',command=mozdit_fel).pack()
Button(abl1,text='Le',command=mozdit_le).pack()

# eseményfigyelő (főhurok) indítása :
abl1.mainloop()
```

A program számos ismert elemet tartalmaz : létrehozunk egy **abl1** ablakot, ebben elhelyezünk egy vásznat, amin egy színes kör, és öt vezérlőgomb van. Vegyük észre, hogy a nyomógomb *widget*-eket nem rendeljük változókhoz (ez felesleges lenne, mert nem fogunk rájuk hivatkozni a későbbiekben). A **pack()** metódust rögtön ezeknek az objektumoknak a létrehozásakor kell alkalmaznunk.

Az igazi újdonság a script elején definiált **mozog()** függvényben van. A függvény minden egyes hívásakor át fogja definiálni a vászonra helyezett « színes kör » objektum koordinátáit és ez idézi elő az objektum animációját.

Ez az eljárás jellemző az « objektum orientált » programozásra: Létrehozunk egy objektumot, majd ***metódusok segítségével módosítjuk a tulajdonságaikat***.

A « régimódi » procedurális (azaz az objektumok használatát nélkülöző) programozásban úgy animáljuk az ábrákat. hogy töröljük őket egy adott helyen, majd egy kicsit távolabb újra rajzoljuk őket. Ezzel szemben az « objektum orientált » programozásban ezeket a feladatokat azok a classok végzik el automatikusan, melyekből az objektumok származnak és ezért nem kell az időnket vesztegetni ezek újraprogramozására.

Gyakorlatok :

8.12. Írjon egy programot, ami megjelenít egy ablakot egy vászonnal. Legyen a vásznon két (különböző méretű és színű) kör, amik két égitestet reprezentálnak. Egy-egy gombbal lehessen őket mozgatni minden irányba. A vászon alatt a programnak ki kell írni: a) a két égitest közötti távolságot; b) a közöttük fellépő gravitációs erőt (Írassuk ki az ablak felső részére az égitestek tömegét valamint a távolságskálát). Ebben a gyakorlatban Newton gravitációs törvényét kell alkalmazni (lásd a 62. oldalon a 42. gyakorlatot és a fizika tankönyvet).

8.13. A vásznon az egérkattintást detektáló program ötletét felhasználva módosítsa a fenti programot úgy, hogy csökkentse a gombok számát : egy égitest elmozdításához legyen elég az égitestet kiválasztani egy gombbal, majd a vászonra kattintani, hogy a kiválasztott égitest arra a helyre kerüljön, ahová kattintott.

8.14. Az előző program bővítése. Jelenítsen meg egy harmadik égitestet és mindíg írassa ki minden egyes égitestre a ható eredők erőjét (mindegyik égitest mindíg a másik kettő által kifejtett gravitációs erő hatása alatt áll !).

8.15. Ugyanaz a gyakorlat, mint az előző, két elektromos töltéssel (Coulomb törvény). Adjon lehetőséget a töltés előjelének megválasztására !

8.16. Írjon egy programot, ami megjelenít egy ablakot két mezővel : az egyik írja ki a hőmérsékletet *Celsius* fokban, a másik ugyanazt a hőfokot *Fahrenheit* fokban. A két hőmérséklet bármelyikét megváltoztatva a másik megfelelően módosuljon. Az átváltáshoz a $T_F = T_C \times 1{,}80 + 32$ képletet használja. Nézze meg az egyszerű kalkulátoros programot is !

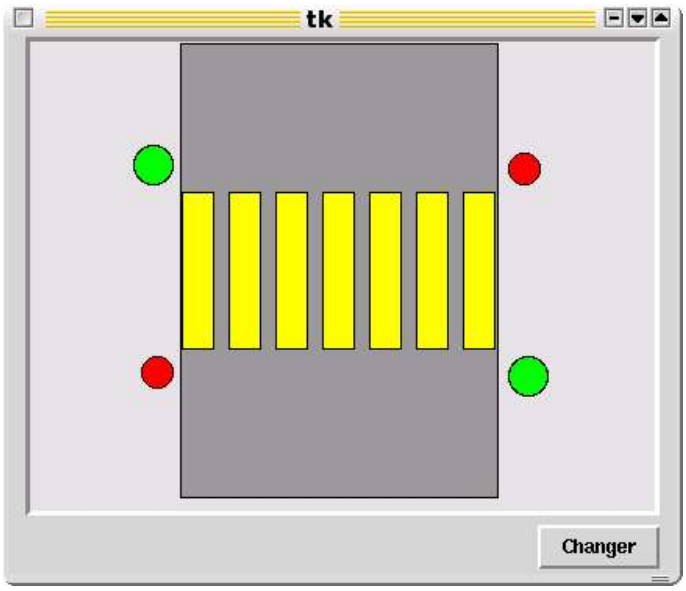
8.17. Írjon egy programot, ami megjelenít egy ablakot egy vászonnal. Helyezzen el ezen a vásznon egy labdát reprezentáló kis kört. Tegyen el a vászon alá egy gombot. A gombon történő minden egyes kattintáskor a labdának egy kis távolsággal jobbra kell elmozdulni mindaddig, amíg el nem éri a vászon szélét. Ha tovább folytatja a kattintgatást a labdának visszafele kell jönni az ellenkező irányba mindaddig, amíg el nem éri a vászon másik szélét és így tovább.

8.18. Módosítsa úgy a programot, hogy a labda egy kör- vagy ellipszispályát járjon be a vásznon (ha folyamatosan kattogtatunk). Megjegyzés : ahhoz, hogy a kiszámolt eredményhez jussunk, egy segédváltozót kell definiálni, ami a leírt szöget fogja reprezentálni és a sinus és cosinus függvényeket kell használni, hogy ennek a szögnek a függvényében pozícionáljuk a labdát.

8.19. Módosítsa úgy az előző programot, hogy a mozgó labda hagyja maga mögött a leírt pálya nyomát.

8.20. Módosítsa úgy az előző programot, hogy a labda más görbéket írjon le. Kérjen ötletet a tanárától (Lissajous-görbék).

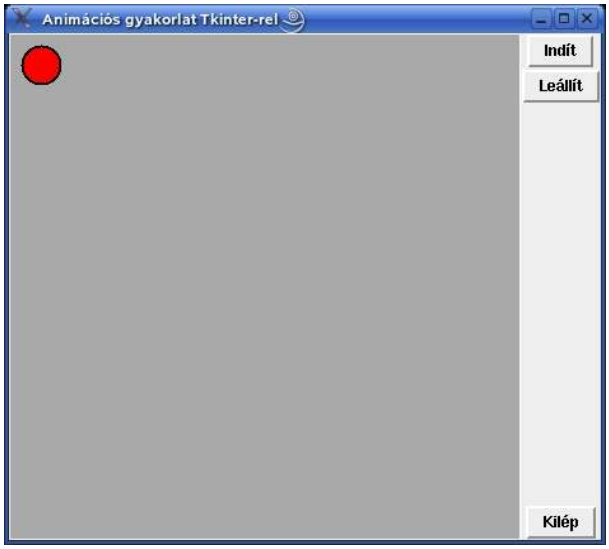
8.21. Írjon egy programot, ami megjelenít egy ablakot egy vászonnal és egy gombbal. A vásznon rajzoljon egy sötétszürke téglalapot, ami egy utat jelöl, fölé pedig sárga téglalapokat , amik egy gyalogátkelőhelyet reprezentálnak. Helyezzen el négy színes kört, amik a gyalogosok és a járművek közlekedési lámpáit reprezentálják. A nyomógombra történő minden egyes kattintásra a lámpáknak színt kell váltani :



8.22. Írjon egy programot, ami egy vásznat jelenít meg, amire egy egyszerű elektromos áramkör van rajzolva (feszültségforrás + kapcsoló + ellenállás). Az ablakban legyenek adatbeviteli mezők, amik lehetővé teszik mindegyik áramköri elem paraméterezését (azaz az ellenállás- és feszültségértékek megválasztását.) A kapcsolónak működőképesnek kell lenni. (Egy « Ki/Be » kapcsológombról gondoskodjunk). A « címkéknek » a felhasználó választásából adódó feszülség és áramerősség értékeket kell megjeleníteni.

8.8 Automatikus animáció - Rekurzivitás

A *Tkinter* grafikus interface-szel való első találkozásunk összegzéseként nézzük meg az utolsó animációs példát, ami ez alkalommal az elindítása után autonóm módon fog működni.




```
from Tkinter import *

# eseménykezelők
# definiálása

def move():
    "a labda elmozdulása"
    global x1, y1, dx, dy, flag
    x1, y1 = x1 +dx, y1 + dy
    if x1 >360:
        x1, dx, dy = 360, 0, 15
    if y1 >360:
        y1, dx, dy = 360, -15, 0
    if x1 <10:
        x1, dx, dy = 10, 0, -15
    if y1 <10:
        y1, dx, dy = 10, 15, 0
    can1.coords(oval1,x1,y1,x1+30,y1+30)
    if flag >0:
        abl1.after(50, move)     # 50 millisec után ciklus

def stop_it():
    "az animáció leáll"
    global flag
    flag =0

def start_it():
    "az animáció elindítása"
    global flag
    flag = flag +1      # flag beállítása az indításhoz :
    if flag ==1:        # magyarázatot lásd aszövegben
        move()

#========== Főprogram =============
# a következő változókat globális változókként fogjuk használni :
x1, y1 = 10, 10         # kezdő koordináták
dx, dy = 15, 0          # nincs elmozdulás
flag =0                 # kapcsoló

# A fő-widget létrehozása ("master") :
abl1 = Tk()
abl1.title("Animációs gyakorlat Tkinter-rel")

# a "slave" widget-ek (canvas + oval, button) létrehozása:
can1 = Canvas(abl1,bg='dark grey',height=400,width=400)
can1.pack(side=LEFT)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
Button(abl1,text='Kilép',command=abl1.quit).pack(side=BOTTOM)
Button(abl1,text='Indít',command=start_it).pack()
Button(abl1,text='Leállít',command=stop_it).pack()

# az eseményfogadó indítása (főciklus) :
abl1.mainloop()
```

A scriptben az egyetlen újdonság a **move()** függvény definíciójának végén található **after()** metódus használata. Ezt a metódust bármilyen ablakra alkalmazhatjuk. Egy függvényhívást fog kezdeményezni ***egy bizonyos idő eltelte után***. Így például a **window.after(200,qqc)** a **window** *widget-en* 200 msec szünet után hívja a **qqc()** függvényt.

Scriptünkben az **after()** metódus önmagát a **move()** függvényt hívja. Most először használjuk a ***rekurzió***nak nevezett igen hatékony programozási technikát. Az egyszerűség kedvéért azt fogjuk mondani, hogy ***akkor történik rekurzió, amikor egy függvény önmagát hívja***. Nyilván így egy végtelen ciklust kapunk, hacsak nem gondoskodunk előre valamilyen eszközről, hogy megszakítsuk azt.

Nézzük, hogyan működik ez a példánkban :

A **move()** függvényt először az « Indít » gombra való kattintáskor hívja a program. A függvény elvégzi a feladatát (vagyis pozícionálja a labdát) majd egy kis szünet után önmagát hívja. Elvégzi feladatát, majd egy kis szünet után újra hívja önmagát és ez így megy tovább a végtelenségig ...

Legalábbis ez történne, ha elővigyázatosságból nem tettünk volna valahová a hurokba egy kilépési utasítást. Egy egyszerű feltételvizsgálatról van szó : minden iterrációs ciklusban megvizsgáljuk egy **if** utasítás segítségével a **flag** változó tartalmát. Ha a **flag** változó tartalma nulla, akkor a ciklus többet nem fut le és az animáció leáll. Mivel a **flag** változó egy globális változó, ezért más függvények segítségével könnyen megváltoztathatjuk az értékét. Azokkal a függvényekkel, amiket az « Indít » és « Leállít » gombokhoz kapcsoltunk.

Így egy egyszerű mechanizmust kapunk az animáció indítására és leállítására :

Az első kattintás az « Indít » gombra egy nem nulla értéket rendel a **flag** változóhoz, majd rögtön kiváltja a **move()** függvény első hívását. Ez végrehajtódik és ez után minden 50 secundumban mindaddig hívja önmagát, amíg a **flag** nullává nem válik. Ha tovább kattintgatunk az « Indít » gombra, a **flag** értéke nő, de semmi egyéb nem történik, mert a **move()** hívására csak akkor kerül sor, ha a **flag** értke 1. Így elkerüljük több, konkurrens ciklus indítását.

A « Leállít » gomb visszaállítja a **flag** értékét nullára és a ciklus megszakad.

Gyakorlatok :

8.23. Törölje a start_it() függvényben az **if flag == 0:** utasítást (és az azt követő két sort). Mi történik ? (Kattintson többször az « Indítás » gombra.)
Próbálja meg elmagyarázni, hogy mi történik.

8.24. Módosítsa úgy a programot, hogy a labda minden fordulónál váltsa a színét..

8.25. Módosítsa úgy a programot, hogy a labda ferde mozgást végezzen, mint amilyet a biliárdasztal széléről visszapattanó biliárdgolyó (« cikk-cakk »).

8.26. Módosítsa úgy a programot, hogy más mozgásokat kapjon. Próbáljon meg pl. körmozgást kapni. (Mint a 105 .oldal gyakorlatában.)

8.27. Módosítsa úgy a programot, vagy írjon egy másik hasonlót, hogy a nehézségi erő hatására eső labda mozgását szimulálja, ami visszapattan a földről. Figyelem : ez esetben gyorsuló mozgásokról van szó !

8.28. Az előző scriptekből mostmár tud írni egy játékprogramot, ami a következő képpen működik : Kis sebességgel véletlenszerűen mozog egy labda a vásznon. A játékosnak meg kell próbálni az egérrel rákattintani a labdára. Ha sikerült, akkor kap egy pontot de a labda ettől kezdve egy kicsit gyorsabban mozog és így tovább. Bizonyos számú kattintás után állítsa le a játékot és írassa ki az elért pontszámot.

8.29. Az előző játék változata : minden alkalommal, amikor a játékosnak sikerült « elkapni », a labda mérete kisebb lesz (a színét is megváltoztathatja).

8.30. Írjon egy programot, amiben több, különböző színű labda mozog, amik egymásról és az oldalfalakról visszapattannak.

8.31. Tökéletesítse az előző gyakorlatok játékait az előbbi algoritmus beépítésével. Most a játékosnak csak a piros labdára kell kattintani. Egy hibás kattintásra (más színű labda eltalálására) pontot veszít.

8.32. Írjon egy programot, ami a Nap körül különböző körpályán keringő égitestek (vagy az atommag körül keringő két elektron) mozgását szimulálja.

8.33. Írjon egy programot, a « kígyójátékra » : egy kígyó (egy négyzetekből álló rövid vonal) a négy irány : föl, le, balra, jobbra valamelyikében mozog a vásznon. A játékos a kígyó haladási irányát a nyílbillentyűkkel bármikor megváltoztathatja. A vásznon « zsákmányok » is találhatók (kis véletlenszerűen elhelyezett, mozdulatlan körök). Úgy kell a kígyót irányítani, hogy « megegye » a zsákmányokat, de ne érjen a vászon széleihez. Amikor a kígyó megeszik egy zsákmányt, egy négyzettel hosszabb lesz és egy új zsákmány jelenik meg valahol. A játék akkor fejeződik be, amikor a kígyó megérinti az egyik falat vagy elér egy bizonyos méretet.

8.34. Az előbbi játék tökéletesítése : a játék akkor is megszakad, ha a kígyó átmetszi önmagát.

9. Fejezet : A file-ok

A programjaink eddig csak nagyon kevés adatot kezeltek. Ezeket minden alkalommal a programtestbe kódolhatnánk (például egy listába). Ez az eljárás azonban alkalmatlan nagyobb tömegű adat kezelésére.

9.1 A file-ok haszna

Az eddigi programjaink csak nagyon kevés adatot kezeltek. Így mindíg kódolni tudtuk őket (például egy listát) a programba. Ez az eljárás azonban alkalmatlan nagymennyiségű adat kezelésére.

Képzeljük pédául el, hogy egy gyakorló programot akarunk írni, ami a képernyőre többválasztásos kérdéseket ír ki és automatikusan kezeli a felhasználó válaszait. Hogyan fogjuk tárolni a kérdések szövegét ?

A legegyszerűbb elképzelés : a program elején értékadó utasításokkal mindegyik szöveget egy változóban tároljuk. Például :

```
a = "Quelle est la capitale du Guatémala ?"
b = "Qui à succédé à Henri IV ?"
c = "Combien font 26 x 43 ?"
     ... etc.
```

Sajnos ez egy nagyon leegyszerűsítő elképzelés. Minden bonyolulttá fog válni, amikor megpróbáljuk majd folytatni a programot, vagyis azokat az utasításokat megírni, amiknek véletlenszerűen kell kiválasztani egyik vagy másik kérdést, hogy azokat kiírják a felhasználónak. Nem biztos, hogy egy olyan hosszú **if ... elif ... elif ...** utasítássorozat, mint ami az alábbi példában van jó megoldás (egyébként elég fárasztó lenne leírni : ne felejtsük el, hogy nagyszámú kérdést akarunk kezelni !) :

```
if choice == 1:
    selection = a
elif choice == 2:
    selection = b
elif choice == 3:
    selection = c
     ... etc.
```

A helyzet sokkal jobbnak tűnik, ha egy listát készítünk :

```
lista= ["Qui a vaincu Napoléon à Waterloo ?",
         "Comment traduit-on 'informatique' en anglais ?",
         "Quelle est la formule chimique du méthane ?", ... etc ...]
```

Az indexe segítségével bármelyik elemet kivehetjük ebből a listából. Példa :

```
print liste[2]        ===>  "Quelle est la formule chimique du méthane ?"
```

(ismétlés : az indexelés nullától kezdődik)

Bár ez az eljárás az előzőnél sokkal jobb, mégis több nem kívánatos problémával találkozunk :

- A program olvashatósága nagyon gyorsan fog romlani, amikor a kérdések száma jelentősen megnő. Ennek következtében megnő a valószínűsége annak, hogy egy vagy több szintaxishibát fogunk ejteni a hosszú listában. Az ilyen hibákat nagyon nehéz kiküszöbölni.
- Új kérdések hozzáadása, vagy bizonyos kérdések módosítása minden alkalommal a program forráskódjának megnyitásával jár. Ennek következtében kényelmetlenné válik ugyanannak a forráskódnak az újra írása, mivel nagyszámú terjedelmes adatsort fog tartalmazni.
- Az adatcsere más programokkal (amiket esetleg más programozási nyelveken írtak) teljesen lehetetlen, mert az adatok a program részét képezik.

Ez utóbbi megjegyzés sugallja a követendő irányt : itt az ideje, hogy megtanuljuk ***külön file-okba szétválasztani az adatokat és az őket kezelő programokat***.

Ahhoz, hogy ez lehetséges legyen, el kell lássuk a programjainkat különböző mechanizmusokkal, amik lehetővé teszik a file-ok létrehozását, azokba adatok mentését, később azok visszanyerését.

A programozási nyelvek többé-kevésbé kifinomult utasításkészletet kínálnak ezeknek a feladatoknak az elvégzéséhez. Amikor az adatok mennyisége nagyon megnő, akkor szükségessé válik a közöttük fennálló kapcsolatok struktúrálása. Ekkor ***relációs adatbázisok***nak nevezett rendszereket kell kidolgozni, amiknek a kezelése nagyon összetett lehet. Ez olyan specializált programoknak a feladata, mint az *Oracle, IBM DB, Adabas, PostgreSQL, MySQL*, stb. A Python tökéletesen alkalmas az ezekkel a rendszer való dialógusra, azonban ezt a későbbiekre hagyom (lásd 252. oldalt).

Jelenlegi ambícióink sokkal szerényebbek. Nincsenek százezres mennyiségű adataink, egyszerű módszerekkel is megelégedhetünk, melyekkel egy közepes méretű file-ba bejegyezhetjük és aztán kinyerhetjük onnan azokat.

9.2 Munkavégzés fileokkal

Egy file használata sokban hasonlít egy könyvéhez. Ahhoz, hogy egy könyvet használjunk, először meg kell azt találni (a címe segítségével), utána ki kell nyitni. Amikor befejeztük a használatát, be kell zárni. Amíg nyitva van különböző információkat lehet belőle olvasni és megjegyzéseket lehet bele írni, de általában a kettőt nem tesszük egyszerre. Az oldalszámok segítségével minden esetben meg tudjuk határozni, hogy hol tartunk a könyvben. A könyvek többségét a lapok normális sorrendjét követve olvassuk, de dönthetünk úgy is, hogy egy tetszőleges bekezdést olvasunk el.

Mindaz, amit a könyvekről elmondtam a file-okra is alkalmazható. Egy file a merev lemezre, floppy discre vagy egy CD-ROM-ra rögzített adatokból áll. A neve segítségével férünk hozzá (ami tartalmazhat egy könyvtárnevet is). A file tartalmát mindíg tekinthetjük egy karaktersorozatnak, ami azt jelenti, hogy ezt a tartalmat, vagy annak bármely részét a karakterláncok kezelésére szolgáló függvények segítségével kezelhetjük.

9.3 Filenevek – Aktuális könyvtár

A magyarázatok egyszerűsítése kedvéért csak a kezelt filok nevét adom meg. A Python a szóban forgó file-okat az ***aktuális könyvtárban*** fogja létrehozni és keresni. Ez szokás szerint az a könyvtár, ahol maga a script található, kivéve ha a scriptet egy *IDLE shell* ablakból indítjuk. Ez esetben az aktuális könyvtár az *IDLE* indításakor van definiálva (*Windows* alatt ennek a könyvtárnak a definíciója részét képezi az indító ikon tulajdonságainak).

Ha az *IDLE*-vel dolgozunk, akkor biztos, hogy a Pythont az aktuális könyvtárának megváltoztatására akarjuk majd kényszeríteni azért, hogy az megfeleljen az elvárásainknak. Ennek megtételéhez használjuk a következő utasításokat a session elején. (Most feltételezem, hogy a használni szándékozott könyvtár a **/home/jules/exercices**). Használhatjuk ezt a szintaxist (vagyis szeparátorként a / karaktert nem pedig a \ karaktert : ez a konvenció érvényes a Unix-világban). A Python automatikusan elvégzi a szükséges konverziókat annak megfelelően, hogy *MacOS*, *Linux*, vagy *Windows*[38] -ban dolgozunk.

```
>>> from os import chdir
>>> chdir("/home/jules/exercices")
```

Az első parancs az **os** modul **chdir()** függvényét importálja. Az **os** modul egy sor függvényt tartalmaz, amik az operációs rendszerrel (os = *operating system*) való kommunikációt teszik lehetővé, bármi is legyen az operációs rendszerünk.

A második parancs a könyvtárat változtatja meg (« chdir » ≡ « *change directory* »).

Megjegyzések :

- Vagy ezeket a parancsokat szúrjuk be a script elejére, vagy pedig megadjuk a file-ok nevében a komplett elérési utat, de ez azzal a veszéllyel jár, hogy megnehezítjük a programjaink írását.
- Részesítsük előnyben a rövid fileneveket. Mindenképpen kerüljük az ékezetes karakterek, a szóközök és a speciális tipográfiai jelek használatát.

9.4 A két import forma

Az imént alkalmazott utasítássorok alkalmat adnak egy érdekes mechanizmus magyarázatára. Tudjuk, hogy az alapmodulba integrált függvények kiegészítéseként a Python nagyszámú, specializáltabb függvényt bocsát a rendelkezésünkre, amik a ***modulok***ban vannak csoportosítva. Már ismerjük a ***math*** és a ***Tkinter*** modulokat.

Ahhoz, hogy egy modul függvényeit használhassuk, importálni kell azokat. Ez kétféle módon történhet, amit a következőkben fogunk megnézni. Mindkét módszernek vannak előnyei és hátrányai.

Egy példa az első módszerre :

```
>>>>>> import os
>>> rep_cour = os.getcwd()
>>> print rep_cour
C:\Python22\essais
```

38 A Windows esetében bele lehet venni az elérési útba annak a tároló perifériának a betűjelét, ahol a file megtalálható. Például : "D:/home/jules/exercices".

A példa első sora ***teljes egészében*** importálja az **os** modult, ami számos függvényt tartalmaz az operációs rendszer eléréséhez. A második sor az **os**[39]modul **getcwd()** függvényét használja. Megállapítható, hogy a **getcwd()** függvény az aktuális könyvtár nevét adja vissza (*getcwd = get current working directory*).

Összehasonlításul íme a másik importálási módot használó példa:

```
>>> from os import getcwd
>>> rep_cour = getcwd()
>>> print rep_cour
C:\Python22\essais
```

Ebben a példában az **os** modulból egyedül a **getcwd()** függvényt importáltuk. Ezen a módon importálva a **getcwd()** függvény úgy épül be a kódunkba, mintha azt mi magunk írtuk volna. Azokban a sorokban, ahol használjuk, nem kell megismételni, hogy az **os** modul része.

Ugyanígy importálhatjuk ugyanannak a modulnak több függvényét :

```
>>> from math import sqrt, pi, sin, cos
>>> print pi
3.14159265359
>>> print sqrt(5)                # négyzetgyök 5
2.2360679775
>>> print sin(pi/6)              # sinus 30°
0.5
```

Sőt a következő módon az összes függvényt importálhatjuk egy modulból :

```
from Tkinter import *
```

Ez az importálási mód megkönnyíti a kódírást. A hátránya az (különösen az utóbbi formának, amelyik egy modul összes függvényét importálja), hogy az aktuális névteret blokkolja. Előfordulhat, hogy bizonyos importált függvények neve megegyezik egy általunk definiált változó nevével, vagy más modulból importált függvények nevével. (Ha ez történik, akkor a két ütköző név egyike nyilvánvalóan már többé nem lesz hozzáférhető).

Az olyan programokban, melyek nagyszámú, különböző eredetű modult hívnak mindíg inkább az első módszert érdemes előnyben részesíteni, vagyis amelyik a minősített neveket használja.

Általában ez alól a szabály alól kivételt teszünk a *Tkinter* modul speciális esetében, mert azokat a függvények, amiket ez a modul tartalmaz nagyon gyakran hívjuk (már ha ennek a modulnak a használata mellett döntünk).

39 A szeparátor pont itt tartalmazási relációt fejez ki. Egy ***minősített névmegadás***i formáról van szó, amit egyre gyakrabban fogunk alkalmazni. Nevek pontokkal való összekapcsolásával egyértelműen jelezzük, hogy az elemek csoportok részeit alkotják, amik maguk még nagyobb csoportok részeit alkothatják, stb. Például a **systeme.machin.truc** címke a **truc** elemet jelöli, ami a **machin** csoport részét képezi, ami maga a **systeme** csoport részét képezi. Erre a jelölési technikára számos példát fogunk látni az objektumosztályok tanulmányozásakor.

9.5 Szekvenciális írás file-ba

A Pythonban a file-okhoz való hozzáférést egy közbenső « ***fileobjektum*** » biztosítja, amit az **open()** belső függvény segítségével hozunk létre. Ennek a függvénynek a hívása után a fileobjektum speciális metódusait használva olvasni és írni tudunk a fileban.

Az alábbi példa bemutatja, hogyan nyitunk meg egy file-t « írásra », jegyzünk be két karakterláncot, majd zárjuk le a file-t. Jegyezzük meg, hogy ha a file még nem létezik, akkor automatikusan létre lesz hozva. Ha viszont a név egy már létező és adatokat tartalmazó file-ra vonatkozik, akkor a bejegyzendő karaktereket hozzá fogja fűzni a meglévőekhez. Ezt a gyakorlatot parancssorból végrehajthatjuk :

```
>>> fileObjektum = open('sajatFile','a')
>>> fileObjektum.write('Bonjour, fichier !')
>>> fileObjektum.write("Quel beau temps, aujourd'hui !")
>>> fileObjektum.close()
>>>
```

Megjegyzések :

- Az első sor egy « fileObjektum » nevű file-objektumot hoz létre. Ez egy valódi file-ra hivatkozik (a merevlemezen vagy a hajlékony lemezen), aminek a neve « sajatFile » lesz. ***Ne keverjük össze a file nevét a file-objektum nevével***, amin keresztül hozzáférünk a filehoz. A gyakorlat során ellenőrizhetjük, hogy a rendszerünkben (az aktuális könyvtárban) egy « sajatFile » nevű file-t (aminek a tartalmát bármelyik szövegszerkesztővel megnézhetjük) hozott létre a program.
- Az **open()** függvény két string típusú argumentumot vár. Az első a megnyitandó file neve, a második a megnyitás módja. Az « **a** » azt jelenti, hogy « hozzáfűzés » (*append*) módban kell a file-t megnyitni, azaz a bejegyzendő adatokat a file végéhez, a már esetleg ott lévő adatokhoz kell fűzni. A « **w** » (*írásra*) filemegnyitási módot is használhattuk volna, de ha ezt a módot használjuk, a Python mindíg egy új (üres) file-t hoz létre és az adatok írása ennek az üres file-nak az elejétől kezdődik. Ha már létezik egy azonos nevű file, akkor azt előbb törli.
- A **write()** metódus végzi a file-ba írást. A kiírandó adatokat argumentumként kell megadni. Ezeket az adatokat sorban egymás után írja ki a file-ba (ezért beszélünk ***szekvenciális*** hozzáférésű file-ról). A **write()** minden új hívása a már rögzített adatok után folytatja az írást.
- A **close()** metódus lezárja a file-t. Ettől kezdve az mindenféle használatra rendelkezésre áll.

9.6 File szekvenciális olvasása

Újra megnyitjuk a file-t, de ez alkalommal « olvasásra », hogy vissza tudjuk olvasni az előző fejezetben rögzített információkat :

```
>>> ofi = open('sajatFile', 'r')
>>> t = ofi.read()
>>> print t
Bonjour, fichier !Quel beau temps, aujourd'hui !
>>> ofi.close()
```

Amint az várható a **read()** metódus kiolvassa a filebeli adatokat és egy « karakterlánc » (*string*) típusú változóba teszi. Ha argumentum nélkül használjuk ezt a metódust, akkor az egész file-tartalmat beolvassa.

Megjegyzések :

- « sajatFile » annak a file-nak a neve, amit olvasni akarunk. A file-t megnyitó utasításnak szükségszerűen hivatkozni kell erre a névre. Ha nem létezik a file, akkor egy hibaüzenetet kapunk. Példa :

```
>>> ofi = open('sajatFile','r')
IOError: [Errno 2] No such file or directory: 'sajatFile'
```

Viszont semmilyen kikötést sem tettünk a file-objektum nevének megválasztására vonatkozóan. Ez egy tetszőleges változónév. Így az első utasításunkban egy « **ofi** »nevű file-objektumot hoztunk létre, ami az olvasásra (« **r** » argumentum) megnyitott « sajatFile » valódi file-ra hivatkozik.

- A két beírt string most egyetlen stringként van a fileban. Ez így természetes, mivel semmilyen elválasztó karaktert sem adtunk meg, amikor beírtuk őket a file-ba. A későbbiekben majd meglátjuk, hogy hogyan kell különálló szövegsorokat rögzíteni.
- A **read()** metódust argumentummal is használhatjuk. Az argumentum azt adja meg, hogy hány karaktert kell beolvasni a file-ban már elért pozíciótól :

```
>>> ofi = open('sajatFile', 'r')
>>> t = ofi.read(7)
>>> print t
Bonjour
>>> t = ofi.read(15)
>>> print t
, fichier !Quel
```

Ha a fileban nincs annyi karakter hátra, mint amennyit az argumentum megad, akkor az olvasás a file végén egyszerűen leáll :

```
>>> t = ofi.read(1000)
>>> print t
 beau temps, aujourd'hui !

Ha a file végén vagyunk, akkor a read() metódus egy üres stringet küld
vissza :
>>> t = ofi.read()
>>> print t


>>> ofi.close()
```

9.7 A ciklusból való kilépésre szolgáló break utasítás

Magától értetődik, hogy programhurkokra van szükségünk, amikor egy olyan file-t kell kezelnünk, aminek nem ismerjük előre a tartalmát. Az alapelképzelés az, hogy részletekben olvassuk a file-t mindaddig, amíg el nem érjük a file végét.

A következő függvény illusztrálja ezt az elképzelést. Az egész file-t – bármekkora is legyen a mérete – átmásolja egy másik file-ba 50 karakteres részletekben :

```
def copyFile(source, destination):
    "filemásolás"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.read(50)
        if txt =="":
            break
        fd.write(txt)
    fs.close()
    fd.close()
    return
```

Ha ellenőrizni akarjuk a függvény működését, két argumentumot kell megadni : az első az eredeti file neve, a második a másolat neve. Példa :

```
copyFile('sajatFile','idegenFile')
```

Megjegyzem a **while** ciklus ebben a függvényben máshogyan van megszerkesztve, mint amilyen formában az előzőekben találkoztunk vele. Tudjuk, hogy a **while** utasítást mindíg egy kiértékelendő feltétel követi. Amíg ez a feltétel igaz, addig fog a **while** utasítást követő utasításblokk végrehajtódni. Itt pedig a kiértékelendő feltételt egy állandó helyettesíti. Azt is tudjuk[40], hogy a Python minden nullától különböző numerikus értéket igaznak tekint.

Az így megalkotott **while** ciklus végtelen ciklus, mert a folytatásának a feltétele mindíg igaz. Ez azonban megszakítható a **break** utasítás hívásával, ami többféle kilépési mechanizmus elhelyezését teszi lehetővé ugyanabba a programhurokba :

```
while <feltétel 1> :
    --- különböző utasítások ---
    if <feltétel 2> :
        break
    --- különböző utasítások ---
    if <feltétel 3>:
        break
    stb.
```

Könnyű belátni, hogy a **copyFile()** függvényünkben a **break** utasítás csak akkor fog végrehajtódni, ha elértük a file végét.

40 Lásd : Egy kifejezés igaz/hamis értéke (58. oldal)

9.8 Szövegfile-ok

Egy szövegfile egy olyan file, ami nyomtatható karaktereket és betűközöket tartalmaz egymást követő sorokba rendezve. A sorokat egy nem nyomtatható speciális karakter a « sorvége karakter »[41] választja el egymástól.

A Pythonnal nagyon könnyen kezelhetők az ilyen fajta file-ok. A következő utasítások egy négysoros szövegfile-t hoznak létre :

```
>>> f = open("szovegfile", "w")
>>> f.write("Ez az elso sor\nEz a masodik sor\n")
>>> f.write("Ez a harmadik sor\nEz a negyedik sor\n")
>>> f.close()
```

Vegyük észre, hogy a stringekbe « \n » sorvége jelzéseket szúrunk be azokra a helyekre, ahol el akarjuk egymástól választani a szövegsorokat. Enélkül a marker nélkül a karaktereket egymás után írnánk ki a file-ba, mint az előző példákban.

Az olvasási műveletek alatt a szövegfile sorai külön-külön nyerhetők vissza. A **readline()** metódus például egyszerre csak egy sort olvas (beleértve a sorvége karaktert is) :

```
>>> f = open('szovegfile','r')
>>> t = f.readline()
>>> print t
Ez az elso sor
>>> print f.readline()
Ez a masodik sor
```

A **readlines()** metódus az összes maradék sort egy stringekből álló listába teszi :

```
>>> t = f.readlines()
>>> print t
['Ez a harmadik sor\012', 'Ez a negyedik sor\012']
>>> f.close()
```

41 Az operációs rendszertől függően a sorvége jelzés kódolása eltérő lehet. Windows alatt például egy két karakteres szekvencia (kocsi vissza és soremelés), a Unix típusú rendszerekben (mint amilyen a Linux) egyetlen soremelés, a MacOS alatt egyetlen kocsi vissza karakter jelzi a sorvégét. Elvileg nem kell ezekkel a különbségekkel foglalkoznunk. Az írásműveletek során a Python az operációs rendszerünkben érvényes konvenciókat alkalmazza. Olvasáskor a Python mindhárom konvenciót korrekten interpretálja (tehát ezeket egyenértékűnek tekinti).

Megjegyzések :

- Fent a lista nyers formában jelenik meg, a karakterláncokat határoló aposztrofokkal és numerikus kód formájú speciális karakterekkel. Természetesen végig mehetünk ezen a listán (például egy **while** ciklus segítségével), hogy kinyerjük belőle az egyes karakterláncokat.
- A **readlines()** metódus lehetővé teszi, hogy egyetlen utasítással ovassunk el egy egész file-t. Ez azonban csak akkor lehetséges, ha az olvasandó file nem túl nagy. (Mivel teljes egészében be fogja másolni egy változóba, vagyis a számítógép operatív memóriájába. ezért a memória méretének megfelelően nagynak kell lenni.) Ha nagy file-okat kell kezelnünk, inkább a **readline ()** metódust használjuk egy programhurokban, ahogyan azt a következő példa mutatja.
- Jól jegyezzük meg, hogy a **readline()** metódus egy karakterláncot ad vissza, míg a **readlines()** metódusa egy listát. A file végén a **readline()** egy üres stringet, míg a **readlines()** egy üres listát ad vissza.

A következő script azt mutatja be, hogyan lehet egy olyan függvényt definiálni, amivel egy szövegfile-on bizonyos feldolgozási műveletet lehet végrehajtani. Jelen esetben arról van szó, hogy úgy másolunk át egy szövegfile-t egy másik file-ba, hogy minden olyan sort kihagyunk, ami **'#** karakterrel kezdődik :

```
def filter(source,destination):
    "filemásolás a jelzett sorok kihagyásával"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.readline()
        if txt =='':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
    return
```

A függvény hívásához két argumentumot kell megadni : az eredeti file nevét és annak a file-nak a nevét, ami a szűrt másolatot fogja tartalmazni. Példa :

```
filter('test.txt', 'test_f.txt')
```

9.9 Különböző változók mentése és visszaállítása

A **write()** metódus argumentumának egy karakterláncnak kell lenni. Amit eddig tanultunk, azzal a tudással csak úgy tudunk más értéktípusokat elmenteni, hogy azokat először stringekké alakítjuk. Ezt a beépített **str()** függvény segítségével tehetjük meg :

```
>>> x = 52
>>> f.write(str(x))
```

Meg fogjuk látni, hogy más lehetőségek is léteznek arra, hogy numerikus értékeket karakterláncokká alakítsunk (e tárgykörben lásd : « Karakterláncok formázása »,130. old.) Igazából nem ez a kérdés. Ha a numerikus értékeket először karakterlánccá alakítjuk és így mentjük el öket, akkor azt kockáztatjuk, hogy többé nem fogjuk tudni azokat helyesen visszaalakítani numerikus értékekké amikor majd a file-t olvassuk. Példa :

```
>>> a = 5
>>> b = 2.83
>>> c = 67
>>> f = open('sajatFile', 'w')
>>> f.write(str(a))
>>> f.write(str(b))
>>> f.write(str(c))
>>> f.close()
>>> f = open('sajatFile', 'r')
>>> print f.read()
52.8367
>>> f.close()
```

Három numerikus értéket mentettünk file-ba. De hogyan fogjuk tudni őket megkülönböztetni a karakterláncban, amikor a file-t olvassuk ? Ez lehetetlen ! Semmi sincs, ami jelezné, hogy a file-ban három vagy egyetlen egy, vagy 2, vagy 4 érték van ...

Az ilyen fajta problémákra többféle megoldás van. Az egyik legjobb megoldás az, hogy importálunk egy specializált Python modult : a pickle[42] modult. A következő képpen használjuk :

```
>>> import pickle
>>> f = open('sajatFile', 'w')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f = open('sajatFile', 'r')
>>> t = pickle.load(f)
>>> print t, type(t)
5 <type 'int'>
>>> t = pickle.load(f)
>>> print t, type(t)
2.83 <type 'float'>
>>> t = pickle.load(f)
>>> print t, type(t)
67 <type 'int'>
>>> f.close()
```

42 Angolul a *pickle* kifejezés jelentése "megőrizni". Azért nevezték el így a modult, mert arra szolgál, hogy az adatokat a típusukat megőrizve mentse el.

Ennek a gyakorlatnak a kedvéért úgy tekintjük, hogy az a, b, c változók ugyanazokat az értékeket tartalmazzák, mint az előző példában. A **pickle** modul **dump()** függvénye két argumentumot vár : az első a mentendő változó neve, a második a file-objektum, amibe menteni fogjuk a változó értékét. A **pickle.load()** függvény a fordított irányú műveletet végzi el, vagyis visszaállít minden változót a típusával.

Könnyen megérthetjük, hogy pontosan mit csinálnak a **pickle** modul függvényei, ha például a **read()** metódus segítségével elvégezzük az eredményfile « klasszikus » olvasását.

9.10 Kivételkezelés. A try – except – else utasítások

A ***kivételek*** (***exceptions***) azok a műveletek, amiket az interpreter vagy a compiler akkor hajt végre, amikor a programvégrehajtás során hibát detektál. Általános szabályként a programvégrehajtás megszakad és egy többé-kevésbé explicit hibaüzenet jelenik meg.
Példa :

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

(Egyéb kiegészítő információk is megjelennek, amik megadják, hogy a script mely részén történt a hiba detektálása, de ezeket most nem reprodukálom.)

A hibaüzenet két részből áll, amit **:** választ el. Elöl van a hiba típusa, utána egy - a hibára vonatkozó - specifikus információ következik.

Számos esetben előre lehet látni, hogy bizonyos hibák léphetnek fel a program egyik vagy másik részében. Ezekbe a programrészekbe beépíthetünk olyan speciális utasításokat, amik csak akkor aktiválódnak, ha ezek a hibák fellépnek. Az olyan magasszintű nyelvekben, mint amilyen a Python, lehetőség van arra, hogy egy felügyelő mechanizmust kössünk egy egész ***utasításcsoporthoz*** és így egyszerűsítsük azoknak a hibáknak a kezelését, melyek ezen utasítások bármelyikében felléphetnek.

Az ilyen mechanizmust általánosan ***kivételkezelő mechanizmus***nak nevezik. A Python kivételkezelő mechanizmusa a **try - except – else** utasításcsoportot használja, ami lehetővé teszi egy hiba elfogását és egy - erre a hibára nézve specifikus - scriptrész végrehajtását. Ez a következő módon működik :

A **try** -t követő utasításblokkot a Python feltételesen hajtja végre. Ha az egyik utasítás végrehajtásakor hiba lép fel, akkor a Python törli a hibás utasítást és helyette az **except** -et követő kódblokkot hajtja végre. Ha semmilyen hiba sem lép fel a **try** utáni utasításokban, akkor az **else** -et követő kódblokkot hajtja végre (ha ez az utasítás jelen van). A program végrehajtása mindegyik esetben a későbbi utasításokkal folytatódhat.

Tekintsünk például egy scriptet, ami arra kéri a felhasználót, hogy adja meg egy file nevét, amit olvasásra kell megnyitni. Nem akarjuk, hogy a program kiakadjon, ha a file nem létezik. Azt akarjuk, hogy írjon ki egy figyelmeztetést és a felhasználó esetleg megpróbálhasson beírni egy másik filenevet.

```
filename = raw_input("Irjon be egy filenvet : ")
try:
    f = open(filename, "r")
except:
    print "A file", filename, "nem létezik"
```

Ha úgy látjuk, hogy ez a fajta teszt alkalmas arra, hogy a program más részein is felhasználjuk, akkor egy függvénybe ágyazhatjuk :

```
def existe(fname):
    try:
        f = open(fname,'r')
        f.close()
        return 1
    except:
        return 0

filename = raw_input("Irjon be egy filenevet : ")
if existe(filename):
    print "Ez a file létezik."
else:
    print "A file", filename, "nem létezik."
```

Az is lehetséges, hogy a **try** -t több **except** blokk kövesse, melyek mindegyike egy specifikus hibatípust kezel, azonban ezt a kiegészítést nem fejtem ki. Amennyiben szükséges, úgy az olvasó nézzen meg egy Python referenciaművet ez ügyben.

(9) Gyakorlatok :

9.1. Írjon egy scriptet, ami lehetővé teszi egy szövegfile kényelmes olvasását. A program először kérje a felhasználótól a file nevét. Ezután ajánlja föl a következő választást : vagy új szövegsorokat rögzít, vagy kiírja a file tartalmát. A felhasználónak be kell tudnia írni az egymást követő szövegsorokat, az <Enter> -t használva a sorok elválasztására. A beírás befejezéseként elég egy üres sort bevinni (vagyis elég magában megnyomni az <Enter> -t) A tartalom kiírásakor a soroknak természetes módon kell egymás után következni (a sorvége kódoknak nem kell megjelenni).

9.2. Tegyük fel, hogy rendelkezésére áll egy szövegfile, ami különböző hosszúságú mondatokat tartalmaz. Írjon egy scriptet, ami megkeresi és kiírja a leghosszabb mondatot.

9.3. Írjon egy scriptet, ami automatikusan létrehoz egy szövegfile-t, ami a 2 – 30 -as szorzótáblákat tartalmazza (mindegyik szorzótábla csak 20 tagot tartalmazzon).

9.4. Írjon egy scriptet, ami úgy másol át egy file-t, hogy a szavak között megháromszorozza a szóközök számát.

9.5. Rendelkezésére áll egy szövegfile, aminek minden sora egy valós típusú numerikus érték reprezentációja (exponens nincs). Például:
14.896
7894.6
123.278
stb.
Írjon egy scriptet, ami ezeket az értékeket egész számra kerekítve egy másik file-ba másolja (a kerekítésnek korrektnek kell lenni).

9.6. Írjon egy scriptet, ami összehasonlítja két file tartalmát és jelzi az első eltérést.

9.7. Az A és B már létező fileokból konstruáljon egy harmadik C filet, ami felváltva tartalmaz egy-egy elemet az A illetve B file-ból. Amikor elérte az egyik eredeti file végét, akkor egészítse ki a C filet a másik file maradék elemeivel.

9.8. Írjon egy scriptet, ami lehetővé teszi egy olyan szövegfile kódolását, melynek minden sora különböző személyek vezetéknevét, keresztnevét, címét, postai irányítószámát és telefonszámát fogja tartalmazni (gondoljon például arra, hogy egy klub tagjairól van szó).

9.9. Írjon egy scriptet, ami az előző gyakorlatban használt file-t másolja át úgy, hogy a személyek születési dátumát és nemét hozzáfűzi (a számítógépnek egyesével kell kiírni a sorokat és a felhasználótól kérni minden egyes kiegészítő adat beírását).

9.10. Tegyük fel, hogy megcsinálta az előző gyakorlatot és most van egy olyan file-ja, ami bizonyos számú személy adatait tartalmazza. Írjon egy scriptet, ami lehetővé teszi, hogy kiszedje ebből a fileból azokat a sorokat, amik egy adott postai irányítószámnak felelnek meg.

9.11. Módosítsa az előző gyakorlat scriptjét úgy, hogy azokat a sorokat találja meg, melyek azoknak a személyeknek felelnek meg, akik nevének kezdő betűje az F és M között van az ABC-ben.

9.12. Írjon függvényeket, amik ugyanazt csinálják, mint a pickle modul függvényei (lásd a 118. oldalt). Ezeknek a függvényeknek lehetővé kell tenniük különböző változók rögzítését egy szövegfileba úgy, hogy azokat szisztematikusan követik a formátumukra vonatkozó információk.

10. Fejezet : Az adatstruktúrák mélyebb tárgyalása

Eddig megelégedtünk az egyszerű műveletekkel. Most nagyobb sebességre kapcsolunk.

A már használt az adatstruktúráknak van néhány olyan jellemzője, amiket még nem ismer az olvasó. Más, összetettebb adatstruktúrák megismerésének is elérkezett az ideje.

10.1 A karakterláncok lényege

Az 5. fejezetben már találkoztunk a karakterláncokkal. A numerikus adatoktól eltérően, amik különálló egységek, a karakterláncok (vagy stringek) egy ***összetett adattípus***t képeznek. Ez alatt egy jól definiált egységet értünk, ami maga kisebb egységek együtteséből áll : ezek a karakterek.

A körülményektől függően egy ilyen összetett adatot hol mint egyetlen objektumot, hol mint elemek rendezett sorozatát fogjuk kezelni. Az utóbbi esetben valószínűleg egyenként akarunk majd hozzáférni az elemeihez.

A karakterláncok a Python-objektumok *szekvenciák*nak nevezett kategóriájának képezik részét, aminek a *listák* és *tuple*-k is részei. A szekvenciákon egy sor hasonló műveletet végezhetünk. Ezek közül már ismerünk párat. A következő fejezetekben le fogok írni néhány újabbat.

10.1.1 Konkatenáció, ismétlés

A stringeket + operátorral lehet **konkatenálni** és a * operátorral lehet **ismételni** :

```
>>> n = 'abc' + 'def'             # konkatenáció
>>> m = 'zut ! ' * 4              # ismétlés
>>> print n, m
abcdef zut ! zut ! zut ! zut !
```

Vegyük észre, hogy a + és * operátorokat összeadásra és szorzásra is használhatjuk, amikor numerikus argumentumokra alkalmazzuk őket. Az egy nagyon érdekes jelenség, hogy ugyanazok az operátorok attól függően, hogy milyen környezetben használjuk őket különbözőképpen működhetnek. A mechanizmust **operator overloading**nak nevezik. Más nyelvekben nem mindíg lehetséges az **operator overloading** : ezért például az összeadásra és konkatenációra különböző szimbólumokat kell használnunk.

10.1.2 Indexelés, kivágás, hossz

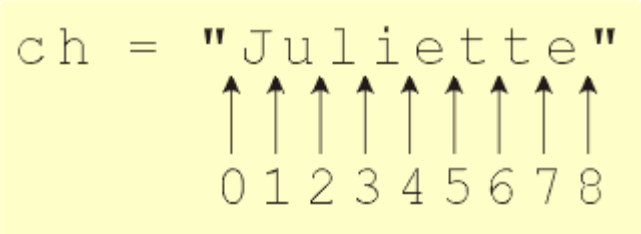
A stringek karaktersorozatok. A szekvenciában mindegyik karakternek pontos helye van. A Pythonban mindíg ugyanúgy indexeljük (sorszámozzuk) egy szekvencia elemeit : ***nullától kezdődően***. Ahhoz, hogy a string egy karakteréhez hozzáférjünk, elég ***szögletes zárójelben*** megadni az indexét :

```
>>> nom = 'Cédric'
>>> print nom[1], nom[3], nom[5]
é r c
```

Amikor egy stringgel dolgozunk, az is nagyon gyakran megtörténik, hogy ki akarunk venni egy hosszabb stringből egy kisebb darabot. A Python erre a « ***slicing*** » (***darabolás/kivágás***) -nak nevezett egyszerű technikát kínálja . Ez abból áll, hogy szögletes zárójelek között meg kell adni a « szelet » elejét és végét, amihez hozzá szeretnénk férni :

```
>>> ch = "Juliette"
>>> print ch[0:3]
Jul
```

Az **[n,m]** szeletbe az n-edik karakter beleértendő, de az m-edik nem. Ha szeretnénk könnyen megjegyezni ezt a mechanizmust, akkor úgy kell elképzelnünk, hogy az indexek a karakterek közötti helyekre mutatnak, mint az alábbi ábrán :



Ezt az elrendezést látva nem nehéz megérteni, hogy a **ch[3:7]** az « **iett** » -et vágja ki.

A vágási indexek alapértelmezett értékei : a definiálatlan első indexet nullának tekinti a Python, míg az elhagyott második index a teljes karakterlánc hosszának értékét veszi föl :

```
>>> print ch[:3]       # az elso 3 karakter
Jul
>>> print ch[3:]       # az elso 3-tol eltekintve az osszes tobbi karakter
iette
```

(10) Gyakorlatok

10.1. Határozza meg, mi történik amikor az egyik vagy a másik vágási index hibás és írja ezt le minél jobban. (Például, ha a második index kisebb, mintaz első, vagy ha a második index nagyobb, mint a string hossza).

10.2. Vágjon föl egy hosszú stringet 5 karakter hosszú darabokra. Rakja össze a darabokat fordított sorrendben.

10.3. Tróbálja megírni a **megtalal()** függvényt, ami az ellenkezőjét csinálja, mint amit az indexoperátor (vagyis a []) tesz. Ahelyett, hogy egy megadott indexhez megtalálja az annak megfelelő karaktert, ennek a függvénynek egy adott karakterhez tartozó indexet kell megtalálni.
Máshogyan fogalmazva, egy olyan függvényt kell írni, ami két argumentumot vár : a kezelendő karakterlánc nevét és a keresendő karaktert. A függvénynek visszatérési értékként az első ilyen karakter stringbeli indexét kell megadni a karakterláncban. Így például : `print megtalal("Juliette & Roméo", "&")` eredménye : **9**

Figyelem : Gondolni kell minden lehetséges esetre. Arra is számítanunk kell, hogy a függvény visszatérési értékként egy speciális értéket (például -1 -et) ad, ha a keresett karakter nincs a karakterláncban.

10.4. Aökéletesítse az előző gyakorlat függvényét úgy, hogy egy harmadik paramétert ad hozzá : azt az indexet, amelyiktől kezdve keresni kell a karakterláncban. Így például a következő utasításnak : **15** -öt kell kiírni (és nem 4 -et !)

```
print megtalal("César & Cléopâtre", "r", 5)
```

10.5. Írjon egy **karakterszam()** függvényt, ami megszámolja, hogy egy karakter hányszor fordul elő egy stringben. Így a következő utasításnak **4** -et kell kiírni :

```
print karakterszam("ananas au jus","a")
```

10.1.3 Szekvencia bejárása. A for ... in ... utasítás

Nagyon gyakran előfordul, hogy egy egész stringet az első karaktertől az utolsóig karakterenként kell kezelnünk azért, hogy mindegyik karakteren valamilyen műveletet végezzünk. Ezt a művleletet ***bejárás***nak fogjuk nevezni. A már ismert **while** utasítás segítségével megírhatjuk egy ilyen bejárás kódját :

```
nev = 'Jacqueline'
index = 0
while index < len(nev):
    print nev[index] + ' *',
    index = index +1
```

A ciklus « bejárja » a **nev** stringet. Egyenként vesz elő minden karaktert, amiket aztán egy csillag közbeiktatásával kinyomtat. Figyeljük meg, hogy a while utasítással alkalmazott feltétel : « **index < len(nev)** ». Ez azt jelenti, hogy a ciklust addig kell futtatni, amíg a 9-es indexértékhez nem érünk (a string 10 karakterből áll). A string valamennyi karakterét kezelni fogjuk, ugyanis azok ***nullától*** 9-ig vannak indexelve.

Egy szekvencia bejárása nagyon gyakori művelet a programozásban.
A Python a **for ... in ... :** utasításpáron alapuló ciklusszerkezetet kínálja a megvalósításához :

A fenti program ezekkel az utasításokkal a következő alakúvá válik :

```
nom = 'Jacqueline'
for caract in nom:
    print caract + ' *',
```

Ez a struktúra tömörebb. Elkerülhető vele egy speciális változó (egy « számláló ») definiálása és incrementálása, amiben az iterrációs lépésben résztvevő karakter indexét tárolnánk. A **caract** változó egymás után fogja tartalmazni a string minden egyes karakterét az elsőtől az utolsóig.

A **for** utasítás tehát olyan ciklusok írását teszi lehetővé, melyekben ***az iterráció az adott szekvencia valamennyi elemét egymás után kezeli***. A fenti példában a szekvencia egy string volt. A következő példa szemlélteti, hogy ugyanezt az eljárást alkalmazhatjuk a listákra (valamint a tuple -kre, amikről a későbbiekben fogunk tanulni) :

```
lista = ['kutya','macska','krokodil']
for allat in lista:
    print allat,'karakterlanc hossza', '=', len(allat)
```

A script végrehajtásának eredménye :

```
kutya karakterlanc hossza= 5
macska karakterlanc hossza= 4
krokodil karakterlanc hossza= 8
```

A **for** utasítás egy új példa az ***összetett utasítás***okra. Ne felejtsük el a kötelező kettőspontot a sor végéről és a rákövetkező blokk behúzását.

Az **in** foglalt szót a kezelendő szekvencia neve követi. A **for** foglalt szót követő név az a név, amit annak a változónak választunk, ami egymás után fogja tartalmazni a szekvencia minden egyes elemét. Ez a változó automatikusan van definiálva (vagyis fölösleges előzetesen definiálni) és típusa automatikusan az éppen kezelt szekvenciaelem típusának felel meg (ismétlés : egy lista esetében nem feltétlenül azonos típusú minden elem).

Példa :

```
divers = ['cheval', 3, 17.25, [5, 'Jean']]
for e in divers:
    print e
```

A script végrahajtásának eredménye :

```
cheval
3
17.25
[5, 'Jean']
```

Bár a **divers** lista elemei mind különböző típusúak (string, egész, valós, lista), egymás után hozzárendelhetjük tartalmukat az **e** változóhoz anélkül, hogy abból hiba származna (ez a Python-változók dinamikus típusadásának köszönhető).

Gyakorlatok :

10.6. Egy amerikai mesében nyolc kiskacsát rendre : Jack, Kack, Lack, Mack, Nack, Oack, Pack és Qack-nak hívnak. Írjon egy scriptet, ami ezeket a neveket a következö két stringből állítja elő :

prefixes = 'JKLMNOP' és suffixe = 'ack'

Ha egy **for ... in ...** utasítást alkalmaz, akkor a scriptjének csak két sort kell tartalmazni.

10.7. Határozza meg egy adott mondatban a szavak számát.

10.1.4 Szekvenciához tartozás. A magában alkalmazott in utasítás

Az **in** utasítás a **for** -tól függetlenül felhasználható annak igazolására, hogy egy adott elem része e egy szekvenciának. Például felhasználhatjuk az **in** -t annak igazolására, hogy valamilyen betű egy meghatározott csoporthoz tartozik-e :

```
car = "e"
maganhangzo = "aeiouyAEIOUY"
if car in maganhangzo:
    print car, "maganhangzo"
```

Hasonló módon lehet igazolni, hogy egy elem egy listához tartozik-e :

```
n = 5
primek = [1, 2, 3, 5, 7, 11, 13, 17]
if  n in primek:
    print n, "resze a primszamlistanknak"
```

Megjegyzés : Ez az igen hatékony utasítás magában a szekvencia tényleges bejárását végzi el. Gyakorlásként írjon olyan utasításokat, amik a **while** utasítást használva egy klasszikus programhurok segítségével ugyanezt a feladatot végeznék el.

Gyakorlatok :

Megjegyzés : a következő feladatokban hagyja figyelmen kívül az ékezetes vagy speciális karaktereket.

10.8. Írjon egy **nagybetu()** függvényt, aminek a visszatérési értéke akkor « igaz », ha az argumentuma nagybetű.

10.9. Írjon egy függvényt, aminek a visszatérési értéke akkor « igaz », ha az argumentuma szám.

10.10. Írjon egy függvényt, ami egy mondatot szavakból álló listává alakít át.

10.11. Használja fel az előző gyakorlatokban definiált függvényeket egy olyan script írására, ami ki tudja szedni egy szövegből az összes nagybetűvel kezdődő szót.

10.1.5 A stringek nem módosítható szekvenciák

Egy létező string tartalmát nem lehet megváltoztatni. Másként fogalmazva, nem használhatjuk a **[]** operátort egy értékadó utasítás baloldalán. Próbáljuk meg például végrehajtani a következő scriptet (ami megpróbál egy betűt kicserélni a stringben) :

```
salut = 'bonjour à tous'
salut[0] = 'B'
print salut
```

A « Bonjour à tous » kiírása helyett a script egy « *TypeError: object doesn't support item assignment* » típusú hibaüzenetet generál. Ezt a hibát a script második sora váltja ki. Abban próbáljuk meg kicserélni a string egyik betűjét egy másikra, de az nem megengedett.

Viszont az alábbi script működik :

```
salut = 'bonjour à tous'
salut = 'B' + salut[1:]
print salut
```

Ebben a másik példában valójában nem a **salut** stringet módosítjuk. Létrehozunk egy uj stringet ugyanazzal a névvel a script második sorában (az előző string egy darabjából, de ami a lényeg : itt egy ***új*** karakterláncról van szó).

10.1.6 A karakterláncok összehasonlíthatók

A karakterláncok esetében is működik mindegyik relációs operátor, amelyikről a programvégrehajtást vezérlő (**if ... elif ... else**) utasításoknál beszéltünk. Ez nagyon hasznos lesz a szavak névsorba rendezésénél :

```
szo = raw_input("Írjon be egy tetszőleges szót : ")
if szo < "limonade":
     place = "megelőzi"
elif szo > "limonade":
     place = "követi"
else:
     place = "fedi"
print "A", szo, "szó", place, "a 'limonade' szót a névsorban"
```

Ezek az összehasonlítások azért lehetségesek, mert a stringeket alkotó alfabetikus karaktereket a számítógép bináris számok formájában tárolja memóriájában, amiknek az értéke a karaktereknek az abc-ben elfoglalt hellyével van összekapcsolva. Az ASCII kódrendszerben például A=65, B=66, C=67, stb.[43]

10.1.7 A karakterek osztályozása

Sokszor hasznos, ha egy karakterláncból kivett karakterről meg tudjuk állapítani, hogy az nagybetű, vagy kisbetű, vagy még általánosabban, ha meg tudjuk határozni, hogy egy számról, vagy más tipográfiai karakterről van szó.

Természetesen tudunk írni olyan függvényeket, amik elvégzik ezt a feladatot. Az első lehetőség az **in** utasítás használata, amint azt az előző fejezetben láttuk. Mivel mostmár tudjuk, hogy a karakterek az ASCII kódban egy rendezett sorozatot alkotnak, ezért fel tudunk használni más módszereket is. Például az alábbi függvénynek akkor « igaz » a visszaérési értéke, ha az argumentuma kisbetű :

```
def kisbetu(ch):
     if 'a' <= ch <= 'z' :
          return 1
     else:
          return 0
```

43 Többféle kódolási rendszer létezik : a legismertebbek az ASCII és az ANSI, melyek egymáshoz elég közeli rendszerek, eltekintve attól ami az angoltól eltérő nyelvek specifikus karaktereit (ékezetes karakterek, cédille, stb.) illeti. Néhány éve a világ valamennyi nyelvének összes speciális karakterét integráló új kódolási rendszer jelent meg. Ezt a unicode-nak nevezett rendszert fokozatosan alkalmazni kell. A 2-es verziójától kezdve ezt integrálja a Python.

Gyakorlatok :

Megjegyzés : a következő gyakorlatokban hagyja figyelmen kívül az ékezetes és speciális karaktereket.

10.12. Írjon egy **nagybetu()** függvényt, aminek akkor « igaz » a visszaérési értéke, ha az argumentuma nagybetű (használjon az előzőekben alkalmazottól eltérő módszert).

10.13. Írjon egy függvényt, aminek akkor « igaz » a visszaérési értéke, ha az argumentuma egy alfabetikus karakter (nagy- vagy kisbetű). Alkalmazza ebben az új függvényben az előzőekben definiált **kisbetu()** és **nagybetu()** függvényeket.

10.14. Írjon egy függvényt, aminek akkor « igaz » a visszaérési értéke, ha az argumentuma egy szám.

10.15. Írjon egy függvényt, ami az argumentumaként megadott mondatban lévő nagybetűk számát adja meg visszaérési értékként.

A Python számos előre definiált függvényt bocsát a rendelkezésünkre, hogy könnyebben tudjunk mindenféle karakterműveletet végezni :

Az **ord(ch)** függvény bármilyen karaktert elfogad argumentumként. Visszatérési értékként a karakternek megfelelő ASCII kódot adja meg. Tehát **ord('A')** visszatérési értéke **65**.

A **chr(num)** függvény ennek pontosan az ellenkezőjét teszi. Az argumentumának 0 és 255 közé eső számnak kell lenni, a 0-t és 255-öt is beleértve. Visszatérési értékként a megfelelő ASCII karaktert kapjuk meg : tehát **chr(65)** az **A** karaktert adja vissza.

Gyakorlatok :

Megjegyzés : a következő gyakorlatokban hagyja figyelmen kívül az ékezetes és speciális karaktereket.

10.16. Írjon egy egy ASCII kódtáblát kiíró scriptet. A programnak minden karaktert ki kell írni. A táblázat alapján állapítsa meg a nagybetűs és kisbetűs karaktereket összekapcsoló relációt minden egyes karakterre.

10.17. Az előző gyakorlatban megtalált reláció alapján írjon egy függvényt, ami egy mondat valamennyi karakterét kisbetűre írja át.

10.18. Ugyanennek a relációnak az alapján írjon egy függvényt, ami valamennyi kisbetűt nagybetűvé alakít át és viszont (az argumentumként megadott mondatban).

10.19. Írjon egy függvényt, ami megszámolja, hogy az argumentumként megadott karakter hányszor fordul elő egy adott mondatban.

10.20. Írjon egy függvényt, ami visszatérési értékként megadja egy adott mondatban a magánhangzók számát.

10.1.8 A karakterláncok objektumok

Az előző fejezetekben már sok ***objektummal*** találkoztunk. Tudjuk, hogy az objektumokon metódusok (vagyis ezekhez az objektumokhoz kapcsolt függvények) segítségével végezhetünk műveleteket. A Pythonban a stringek objektumok. A megfelelő metódusok használatával számos műveletet végezhetünk rajtuk. Íme néhány a leghasznosabbak közül[44] :

- **split() :** egy stringet alakít át substringek listájává. Mi adhatjuk meg argumentumként a szeparátor karaktert. Ha nem adunk meg semmit sem, akkor az alapértelmezett argumentumérték egy szóköz :

```
>>> c2 ="Votez pour moi"
>>> a = c2.split()
>>> print a
['Votez', 'pour', 'moi']
>>> c4 ="Cet exemple, parmi d'autres, peut encore servir"
>>> c4.split(",")
['Cet exemple', " parmi d'autres", ' peut encore servir']
```

- **join(liste) :** egyetlen karakterlánccá egyesít egy stringlistát. (Ez a metódus az előző inverze.) Figyelem : a szeparátor karaktert (egy vagy több karaktert) az a string fogja megadni, amelyikre a metódust alkalmazzuk, az argumentuma az egyesítendő stringek listája :

```
>>> b2 = ["Salut","les","copains"]
>>> print  " ".join(b2)
Salut les copains
>>> print  "---".join(b2)
Salut---les---copains
```

- **find(sch) :** megkeresi az **sch** substring pozícióját egy stringben :

```
>>> ch1 = "Cette leçon vaut bien un sajt, sans doute ?"
>>> ch2 = "sajt"
>>> print ch1.find(ch2)
25
```

- **count(sch) :** megszámolja a **sch** substring előfordulásainak számát a stringben :

```
>>> ch1 = "Le héron au long bec emmanché d'un long cou"
>>> ch2 = 'long'
>>> print ch1.count(ch2)
2
```

- **lower() :** kisbetűssé alakít egy stringet :

>>> ch ="ATTENTION : Danger !"
>>> print ch.lower()
attention : danger !

44 Csak néhány példáról van szó. Ezeknek a metódusoknak a többsége más paraméterekkel is használható, amiket nem mind adok itt meg (például bizonyos paraméterek lehetővé teszik, hogy egy stringnek csak bizonyos részét kezeljük). A **dir()** beépített függvény segítségével megkaphatjuk az objektumhoz kapcsolódó valamennyi metódus teljes listáját. Aki többet akar tudni, nézzen meg egy referencia művet (vagy az on-line dokumentációt).

- **upper() :** nagybetűssé alakítja a karakterláncot :

```
>>> ch = "Merci beaucoup"
>>> print ch.upper()
MERCI BEAUCOUP
```

- **capitalize() :** a karakterlánc első betűjét nagybetűvé alakítja :

```
>>> b3 = "quel beau temps, aujourd'hui !"
>>> print b3.capitalize()
"Quel beau temps, aujourd'hui !"
```

- **swapcase() :** minden nagybetűt kisbetűvé alakít és visszont :

```
>>> ch5 = "La CIGALE et la FOURMI"
>>> print ch5.swapcase()
lA cigale ET LA fourmi
```

- **strip() :** eltávolítja a string elején és végén lévő betűközöket :

```
>>> ch = "   Monty Python   "
>>> ch.strip()
'Monty Python'
```

- **replace(c1, c2) :** A stringben valamennyi **c1** karaktert **c2** -vel helyettesíti :

```
>>> ch8 = "Si ce n'est toi c'est donc ton frère"
>>> print ch8.replace(" ","*")
Si*ce*n'est*toi*c'est*donc*ton*frère
```

index(c) : megadja a **c** karakter stringbeli első előfordulásának indexét :

```
>>> ch9 ="Portez ce vieux whisky au juge blond qui fume"
>>> print ch9.index("w")
16
```

Ezeknek a metódusoknak a többségében kiegészítő argumentumokkal pontosan meg lehet adni, hogy a karakterlánc melyik részét kell kezelni. Példa :

```
>>> print ch9.index("e")          # a string elejetől keresi
4                                 # és megtalálja az első 'e'-t
>>> print ch9.index("e",5)        # csak az 5-os indextől keres
8                                 # és megtalálja a masodik 'e'-t
>>> print ch9.index("e",15)       # a 15-dik karaktertől keres
29                                # és megtalálja a negyedik 'e'-t
```

stb., stb.

- E kurzus keretében nem lehet az összes metódust és a paraméterezésüket leírni. Ha az olvasó többet akar tudni, akkor az online Python dokumentációt (Library reference), vagy egy jó referenciaművet kell megnézni (mint például Fredrik Lundh « Python Standard Library »-jét – Editions O'Reilly) .

Beépített függvények

A karakterláncokra számos olyan függvényt alkalmazhatunk, amik magába a nyelvbe vannak beépítve :

- **len(ch)** megadja **ch** string hosszát (vagyis a karaktereinek a számát)
- **float(ch)** valós számmá (*float*) alakítja a **ch** stringet (természetesen ez csak akkor fog működni, ha a karakterlánc egy ilyen számot reprezentál) :

```
>>> a = float("12.36")
>>> print a + 5
17.36
```

- **int(ch)** egész számmá alakítja a **ch** stringetr :

```
>>> a = int("184")
>>> print a + 20
204
```

10.1.9 Karakterláncok formázása

A karakterláncokkal kapcsolatos függvények világában tett kalandozásunk befejezéseként hasznosnak tűnik még a ***formázás***nak nevezett technika bemutatása. Ez a technika minden olyan esetben nagyon hasznos, amikor különböző változók értékeiből kell egy komplex karakterláncot konstruálni.

Tegyük fel például, hogy írtunk egy programot, ami egy vizes oldat színét és hőmérsékletét kezeli. A színt egy **szin** nevű változóban tároljuk és a hőmérsékletet egy **homers** nevű változóban (*float* típusú változó). Azt akarjuk, hogy a programunk egy új karakterláncot hozzon létre ezekből az adatokból. Például egy ilyen mondatot : « Az oldat színe pirossá változott és hőmérséklete eléri a 12,7 °C-t ».

Ezt a karakterláncot a konkatenáció operátor (a + szimbólum) segítségével összeállíthatjuk részekből, de a **str()** függvényt is használnunk kellene, hogy a *float* típusú változóban tárolt numerikus értéket stringgé alakítsuk (Csinálja meg a gyakorlatot !).

A Python egy másik lehetőséget is kínál. A karakterlánc a % operátor segítségével előállítható két elemből : a baloldalon egy formázó stringet adunk meg (valamilyen mintát) ami konverziós markereket tartalmaz és jobboldalon (zárójelben) egy vagy több objektumot ami(ke)t a Pythonnak a karakterláncba a markerek helyére kell be szúrni.

Példa :

```
>>> szin ="zöld"
>>> homers = 1.347 + 15.9
>>> print "A színe %s és a hömérséklete %s °C" % (szin,homers)
A színe zöld és a hőmérséklete 17.247 °C
```

Ebben a példában a formázó string két `%s` konverziós markert tartalmaz, amiket a Python a **szin** és a **homers** változók tartalmával helyettesít.

A `%s` marker bármilyen objektumot elfogad (stringet, egészet, float-ot, ...). Más markereket alkalmazva más formázással lehet kísérletezni. Próbáljuk meg például a második markert `%s` -t `%d`–vel helyettesíteni, vagy például `%8.2g` -vel. A `%d` marker egy számot vár és azt egésszé alakítja át; a `%f` és `%g` markerek valós számokat várnak és meg tudják határozni a kiírás szélességét és pontosságát.

Az összes formázási lehetőség leírása meghaladja e könyv kereteit. Ha egy speciális formázásra van szüksége, nézze meg a Python online dokumentációját, vagy egy speciális kézikönyvet.

Gyakorlatok :

10.21. Írjon egy scriptet, ami megszámolja egy szövegfileban a numerikus karaktereket tartalmazó sorokat.

10.22. Írjon egy scriptet, ami egy szövegfileban megszámolja a szavakat.

10.23. Írjon egy scriptet, ami átmásol egy szövegfile-t, miközben ügyel arra, hogy minden sor nagybetűvel kezdődjön.

10.24. Írjon egy scriptet, ami úgy másol át egy szövegfilet, hogy azokat a sorokat, amik kisbetűvel kezdődnek összeolvasztja az előző sorral.

10.25. Van egy numerikus értékeket tartalmazó file. Tegyük fel, hogy ezek az értékek egy gömbsorozat átmérői. Írjon egy scriptet, ami arra használja föl ennek a file-nak az adatait, hogy egy másik szövegsorokba rendezett filet hoz létre, ami ezeknek a gömböknek más jellemzőit fejezi ki (főkör területe, felület, térfogat) a következő formában :

```
d   46.20 cm Ft =  1676.39 cm² Fel. =  6705.54 cm². Tf. =  51632.67 cm³
d. 120.00 cm Ft = 11309.73 cm² Fel. = 45238.93 cm². Tf. = 904778.68 cm³
d.   0.03 cm Ft =     0.00 cm² Fel. =     0.00 cm². Tf. =      0.00 cm³
d.  13.90 cm Ft =   151.75 cm² Fel. =   606.99 cm². Tf. =   1406.19 cm³
d.  88.80 cm FT =  6193.21 cm² Fel. = 24772.84 cm². Tf. = 366638.04 cm³
stb.
```

10.26. Van egy szövegfile-ja, aminek a sorai valós típusú (exponens nélküli) numerikus értékeket reprezentálnak (és string formájában vannak kódolva). Írjon egy scriptet, ami átmásolja ezeket az értékeket egy másik fileba úgy, hogy közben a decimális részüket úgy kerekíti, hogy az csak egy tizedes jegyet tartalmaz a tizedes vessző után (a kerekítésnek korrektnek kell lenni).

10.2 A listák lényege

Az 5. fejezetbeli rövid bemutatásuk óta már többször találkoztunk a listákkal. A listák rendezett objektumok gyűjteményei. A stringekhez hasonlóan a ***szekvenciák***nak nevezett általános adattípus részei. Ahogyan egy stringben a karaktereket, úgy a listában elhelyezett objektumokat is egy index-szel érhetjük el (ez egy szám, ami az objektum szekvenciabeli helyét adja meg).

10.2.1 Egy lista definíciója Hozzáférés az elemeihez

Már tudjuk, hogy a listát szögletes zárójellel határoljuk :

```
>>> szamok = [5, 38, 10, 25]
>>> szavak = ["sonka", "sajt", "lekvár", "csokoládé"]
>>> stuff = [5000, "Brigitte", 3.1416, ["Albert", "René", 1947]]
```

Az utolsó példában egy egészet, egy stringet, egy valós számot és egy listát gyűjtöttem össze, hogy emlékeztessek rá : egy listában bármilyen adattípusokat kombinálhatunk, beleértve listákat, tuple-ket és szótárakat (ezekről később fogunk tanulni).

Egy lista elemeihez ugyanazokkal a módszerekkel (index, részekre vágás) férünk hozzá, mint amikkel a stringek karaktereihez :

```
>>> print szamok[2]
10
>>> print szamok[1:3]
[38, 10]
>>> print szamok[2:3]
[10]
>>> print szamok[2:]
[10, 25]
>>> print szamok[:2]
[5, 38]
>>> print szamok[-1]
25
>>> print szamok[-2]
10
```

A fenti példák felhívják a figyelmet arra a tényre, hogy egy listából ***kivágott rész*** maga is mindíg lista (még ha egy olyan részről is van szó, ami csak egy elemet tartalmaz, mint a harmadik példában), az izolált elem pedig bármilyen adattípust tartalmazhat. Ezt a megkülönböztetést a következő példák során el fogjuk mélyíteni.

10.2.2 A listák módosíthatók

A stringekkel szemben a listák ***módosítható*** szekvenciák. Ez lehetővé teszi, hogy a későbbiekben dinamikusan (azaz valamilyen algoritmussal) hozzunk létre részekből nagyméretű listákat.

Példák :

```
>>> szamok[0] = 17
>>>  szamok
[17, 38, 10, 25]
```

A fenti példában az egyenlőségjel baloldalán alkalmazva a **[ ]** operátort (*index operátort*) a **szamok** lista első elemét helyettesítettük más elemmel.

Ha hozzá akarunk férni egy listaelemhez, ami maga egy másik listában van, elég ha a két indexet egymás után lévő szögletes zárójelekbe tesszük :

```
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1415999999999999, ['Albert', 'Isabelle', 1947]]
```

Minden szekvenciára igaz, hogy az elemek számozása nullával kezdődik. Így a fenti példában egy lista 1-es számú elemét helyettesítjük, mely lista egy másik listának (a **stuff** -nak) a 3-as számú eleme.

10.2.3 A listák objektumok

A Pythonban a listák valódi objektumok és így különösen hatékony ***metódusokat*** alkalmazhatunk rájuk :

```
>>> szamok = [17, 38, 10, 25, 72]
>>> szamok.sort()                    # listarendezés
>>>  szamok
[10, 17, 25, 38, 72]
>>> szamok.append(12)                # hozzáfűz egy elemet a végéhez
>>> nombres
[10, 17, 25, 38, 72, 12]
>>> szamok.reverse()                 # az elemek sorrendjének megfordítása
>>> szamok
[12, 72, 38, 25, 17, 10]
>>> szamok.index(17)                 # elem indexének meghatározása
4
>>> szamok.remove(38)                # elem törlése
>>> szamok
[12, 72, 25, 17, 10]
```

Ezeken a metódusokon túl még rendelkezésünkre áll a beépített **del** utasítás, amivel az indexe alapján egy vagy több elemet törölhetünk :

```
>>> del szamok[2]
>>> szamok
[12, 72, 17, 10]
>>> del szamok[1:3]
>>> szamok
[12, 10]
```

Jól jegyezzük meg a **remove()** és a **del** utasítások közötti különbséget : a **del** egy index-szel vagy egy indextartománnyal működik, míg a **remove()** egy ***értéket*** keres (ha a listában több elemnek azonos az értéke, akkor az elsőt törli).

Gyakorlatok :

10.27. Írjon egy scriptet, ami a 20 -tól 40 -ig terjedő számok négyzeteit és köbeit állítja elő.

10.28. Írjon egy scriptet, ami 5°-onként automatikusan előállítja a 0° és 90° közé eső szögek sinusait. Figyelem : a **math** modul **sin()** függvénye úgy tekinti, hogy a szögek radiánban vannak megadva (360° = 2 π radián)

10.29. Írjon egy scriptet, ami a 2, 3, 5, 7, 11, 13, 17, 19 -es (ezeket a számokat egy listába fogjuk elhelyezni) szorzótáblák első 15 tagját állítják elő a képernyőn a következő táblázathoz hasonlóan :

```
 2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
 3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
 5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
 stb.
```

10.30. Legyen adott a következő lista :
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien', 'Alexandre-Benoît', 'Louise']
Írjon egy scriptet, ami kiírja a neveket és a nevekben lévő karakterek számát.

10.31. Van egy egész számokat tartalmazó lista, amiben egyes számok többször is előfordulnak. Írjon egy scriptet, ami a listát úgy másolja át egy másik listába, hogy figyelmen kívül hagyja a többszöri előfordulásokat. A végső listának rendezettnek kell lenni.

10.32. Írjon egy scriptet, ami megkeresi egy adott mondatban a leghosszabb szót (a program felhasználójának be kell tudnia írni egy általa választott mondatot).

10.33. Írjon egy scriptet, ami kiíratja egy csütörtöki nappal kezdődő képzeletbeli év napjainak a listáját. A scriptben három lista lesz : az egyik a hét napjainak a neveit fogja tartalmazni, a másik a hónapok neveit, a harmadik pedig hogy hány naposak a hónapok (a szökőéveket nem vesszük figyelembe).
Példa :
Január 1 csütörtök Január 2 péntek Január 3 szombat Január 4 vasárnap
... és így tovább December 31 csütörtök -ig.

10.34. Van egy file, amiben tanulók nevei találhatók. Írjon egy scriptet, ami rendezetten másolja át ezt a filet.

10.35. Írjon egy függvényt, amivel rendezni lehet egy listát. A függvény ne használja a Python beépített sort() metódusát : tehát egy rendezési algoritmust kell definiálni.

(Megjegyzés : ezt a kérdést az osztályban kell megbeszélni)

10.2.4 Lista módosítására szolgáló haladó « slicing » (szeletelési) technikák

Amint említettem, egy beépített utasítással (**del**) törölhetünk egy elemet egy listából, illetve egy beépített metódussal (**append**()) hozzáfűzhetünk egy elemet a listához. Ha jól elsajátítottuk a « szeletelés » (*slicing*) elvét, akkor a **[ ]** operátorral ugyanezt az eredményt kaphatjuk. Ennek az operátornak a használata egy kicsit nehezebb, mint az erre a feladatra szolgáló utasításoké vagy metódusoké, de több rugalmasságot enged meg :

a) Egy vagy több elem beszúrása egy lista tetszőleges helyére

```
>>> szavak = ['sonka', 'sajt', 'lekvár', 'csokoládé']
>>> szavak[2:2] =["méz"]
>>> szavak
['sonka', 'sajt', 'méz', 'lekvár', 'csokoládé']
>>> szavak[5:5] =['kolbász', 'ketchup']
>>> szavak
['sonka', 'sajt', 'méz', 'lekvár', 'csokoládé', 'kolbász', 'ketchup']
```

Ennek a technikának a használatához a következő sajátosságokat kell figyelembe venni :

- Ha a **[ ]** operátort az egyenlőségjel baloldalán használjuk, hogy eleme(ke)t szúrjunk be a listába vagy töröljünk a listából, akkor kötelező megadni a céllista egy « szeletét » (azaz két indexet a zárójelben); nem pedig egy elszigetelt elemet a listában.
- Az egyenlőségjel jobboldalán megadott elemnek magának is egy listának kell lenni. Ha csak egy elemet szúrunk be, akkor azt szögletes zárójelek közé kell tenni, hogy először egy egyelemű listává alakítsuk. Jegyezzük meg, hogy a **szavak[1]** elem nem lista (ez a « sajt » karakterlánc), míg a **szavak[1:3]** elem egy lista.

A most következők elemzésével jobban megfogjuk érteni ezeket a korlátozásokat :

b) Elemek törlése / helyettesítése

```
>>> szavak[2:5] = []                          # [] üres listát jelöl
>>> szavak
['sonka','sajt','kolbász', 'ketchup']
>>> szavak[1:3] = ['saláta']
>>> szavak
['sonka', 'saláta', 'ketchup']
>>> szavak[1:] = ['majonéz', 'csirke', 'paradicsom']
>>> szavak
['sonka', 'majonéz', 'csirke', 'paradicsom']
```

- A példa első sorában a [2:5] szeletet egy üres listával helyettesítjük, ami egy törlésnek felel meg.
- A negyedik sorban egyetlen elemmel helyettesítünk egy szeletet. (Mégegyszer jegyezzük meg, hogy ennek az elemnek lista formájában kell megjelenni).
- A 7-ik sorban egy kételemű szeletet egy három elemű listával helyettesítünk.

10.2.5 Számokból álló lista létrehozása a range() függvénnyel

Ha számsorokat kell kezelnünk, akkor azokat nagyon könnyen létrehozhatjuk a következő függvénnyel :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A **range()** függvény növekvő értékű egész számokból álló listát generál. Ha a **range()** -et egyetlen argumentummal hívjuk, akkor a lista az argumentummal megegyező számú értéket fog tartalmazni, de az értékek nullától fognak kezdődni (*vagyis a **range(n)** a **0**-tól **n-1** -g terjedő egész számokat hozza létre).*
Jegyezzük meg, hogy a megadott argumentum soha sem szerepel a létrehozott listában.

A **range()**-et két vagy három, vesszővel elválasztott argumentummal is használhatjuk speciálisabb számsorok létrehozására :

```
>>> range(5,13)
[5, 6, 7, 8, 9, 10, 11, 12]
>>> range(3,16,3)
[3, 6, 9, 12, 15]
```

Ha az olvasó nehezen fogadja el a fenti példát, akkor gondoljon arra, hogy a **range()** mindíg három argumentumot vár, amiket *FROM*, *TO* & *STEP* -nek nevezhetnénk. A *FROM* az első érték, amit létre kell hozni, a *TO* az utolsó (vagy inkább az utolsó+1), és a *STEP* a növekmény a következő értékig. Ha nem adjuk meg a *FROM* és *STEP* paramétereket, akkor azok a 0 és 1 értéket veszik fel alapértelmezetten.

10.2.6 Lista bejárása a for, range() és len() segítségével

A **for** utasítás ideális egy lista bejárásához :

```
>>> mondat =
['La','raison','du','plus','fort','est','toujours','la','meilleure']
>>> for szo in mondat:
        print szo,
La raison du plus fort est toujours la meilleure
```

Egy szekvencia (lista vagy karakterlánc) indexeinek automatikus előállításához nagyon praktikus a **range()** és a **len()** függvények kombinálása. Példa :

```
mese = ['Maître','Corbeau','sur','un','arbre','perché']
for index in range(len(mese)):
    print index, mese[index]
```

A script végrehajtásának eredménye :

```
0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

10.2.7 A dinamikus típusadás egy következménye

Amint már föntebb jeleztem (a 124.oldalon), a for utasítással használt változó típusa a bejárás folyamán folyamatosan újra van definiálva. Még akkor is, ha egy lista elemei különböző típusúak, a **for** segítségével úgy járhatjuk be ezt a listát, hogy nem kapunk hibaüzenetet, mert a ciklusváltozó típusa automatikusan alkalmazkodik az adat olvasása során annak típusához. Példa :

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers:
        print item, type(item)

3 <type 'int'>
17.25 <type 'float'>
[5, 'Jean'] <type 'list'>
Linux is not Windoze <type 'str'>
```

A fönti példában a beépített **type()** függvényt arra használom, hogy megmutassam az **item** változó tényleg megváltoztatja a típusát minden iterrációban (ezt a Python dinamikus típusadása teszi lehetővé)

10.2.8 Műveletek listákon

A + (konkatenáció) és * (szorzás) operátorokat alkalmazhatjuk a listákra :

```
>>> gyumolcsok = ['narancs','citrom']
>>> zoldsegek = ['póréhagyma','hagyma','paradicsom']
>>> fruits + legumes
['narancs','citrom','póréhagyma','hagyma','paradicsom']
>>> fruits * 3
['narancs','citrom','narancs','citrom','narancs','citrom']
```

A * operátor különösen hasznos n azonos elemből álló lista létrehozásakor :

```
>>> het_nulla = [0]*7
>>> het_nulla
[0, 0, 0, 0, 0, 0, 0]
```

Például tételezzük fel, hogy egy olyan **B** listát akarunk létrehozni, ami ugyanannyi elemet tartalmaz, mint egy másik **A** lista. Ezt különböző módon érhetjük el, de az egyik legegyszerűbb megoldás a : `B = [0]*len(A)`

10.2.9 Tartalmazás igazolása

Az **in** utasítás segítségével könnyen meghatározhatjuk, hogy egy elem része e egy listának :

```
>>> v = 'paradicsom'
>>> if v in zoldsegek:
        print 'OK'
OK
```

10.2.10 Lista másolása

Tegyük fel, hogy van egy **fable** nevű listánk, amit át szeretnénk másolni a **phrase** nevű új változóba. Az olvasó első ötlete bizonyára egy egyszerű értékadás :

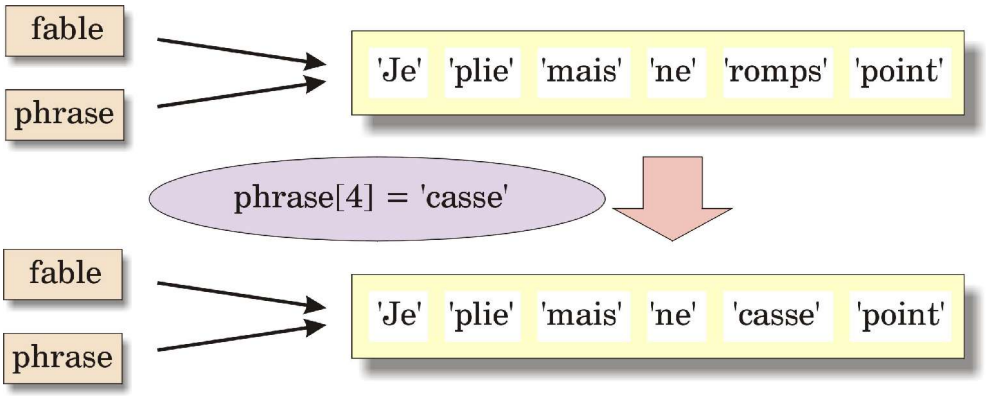
```
>>> phrase = fable
```

Ha így járunk el, nem hozunk létre valódi másolatot. Az utasítást követően még mindíg csak egy változó van a számítógép memóriájában. Amit létrehoztunk az csak ***egy új hivatkozás a listára***. Próbáljuk ki például :

```
>>> fable = ['Je','plie','mais','ne','romps','point']
>>> phrase = fable
>>> fable[4] ='casse'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Ha a **phrase** változó valóban a lista másolatát tartalmazná, akkor az a másolat független lenne az eredetitől. Akkor viszont nem tudná módosítani egy olyan utasítás, mint amit a harmadik sorban a **fable** változóra alkalmaztunk. Kísérletezhetünk még akár a **fable,** akár a **phrase** tartalmának más módosításaival is. Mindegyik esetben meg állapíthatjuk, hogy az egyik lista módosítása tükröződni fog a másik listában és viszont.

Valójában mind a két név ugyanazt az objektumot jelöli a memóriában. Erre a helyzetre az informatikusok azt mondják, hogy a **phrase** név a **fable** név ***aliasa***.



A későbbiekben látni fogjuk az *aliasok* hasznát. Most azt szeretnénk, ha volna egy olyan eljárásunk, amivel létre tudunk hozni egy valódi listamásolatot. Az előzőekben bemutatott fogalmak segítségével az olvasónak magának is tudnia kell találni ilyet.

Megjegyzés :

A Python megengedi, hogy egy hosszú utasítás több sorra terjedjen ki, ha azt (), [], {} zárójelek határolják. Így kezelni tudjuk a zárójeles kifejezéseket, illetve a hosszú lista- , tuple-, illetve szótárdefiníciókat (lásd később). A behúzás mértékének nincs jelentősége : az interpreter ott érzékeli az utasítás végét, ahol a zárójel záródik.

Ez a tulajdonság lehetővé teszi a program olvashatóságának a javítását. Példa :

```
szinek = ['fekete', 'barna', 'piros',
          'narancs', 'sárga', 'zöld',
          'kék', 'ibolya', 'szürke', 'fehér']
```

Gyakorlatok :

10.36. Legyenek adottak a következő listák :

```
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Január','Február','Március','Április','Május','Június',
 'Július','Augusztus','Szeptember','Október','November','December']
```

Írjunk egy kis programot, ami a második listába úgy szúrja be az első lista összes elemét, hogy minegyik hónap nevét az illető hónap napjainak száma követi : `['Január',31,'Február',28,'Március',31, stb...]`.

10.37. Hozzunk létre egy **A** listát, ami tartalmaz néhány elemet. Hozzuk létre ennek egy valódi másolatát az új **B** változóban. Ötlet : először hozzunk létre egy csupa nullákat tartalmazó **B** listát, aminek a mérete megegyezik az **A** lista méretével. Ezután helyettesítsük a nullákat az **A** elemeivel.

10.38. Ugyanaz a probléma, de más az ötlet : először hozzunk létre egy üres **B** listát. Ez után töltsük azt fel az A elemeinek segítségével.

10.39. Ugyanaz a probléma, de megint más az ötlet : a **B** lista létrehozásához az **A** listában vágjunk egy olyan szeletet, ami az összes elemet tartalmazza (a **[:]** operátor segítségével.)

10.40. Egy ***primszám*** olyan szám, ami csak eggyel és önmagával osztható. Írjon programot, ami az ***eratosztenészi szita*** módszerének alkalmazásával az összes 1 és 1000 közötti prímszámot előállítja :

Hozzon létre egy 1000 elemből álló listát, minden listaelem kezdőértéke 1 legyen.

Járja be ezt a listát a 2 indexű elemtől kezdve :
- ha a vizsgált elem értéke 1, akkor tegye nullává a lista azon elemeit, melyeknek indexe egész számú többszöröse az aktuális indexnek
- amikor így bejárta az egész listát, akkor azoknak az elemeknek az indexei lesznek a keresett prímszámok, mely elemek értéke 1 maradt.

Valóbant : A 2 -es indextől kezdve minden elemet nullázunk, aminek az indexe kettő egész számú többszöröse : 4, 6, 8, 10, stb. A 3-as indexxel nullázzuk a 6, 9, 12, 15, stb., indexű elemeket és így tovább.
Csak azoknak az elemeknek marad 1 az értéke amiknek az indexe prímszám.

10.2.11 Véletlenszámok - Hisztogrammok

A számítógép programok többsége pontosan ugyanazt a dolgot csinálja minden futtatáskor. Az ilyen programokat ***determinisztikusaknak*** nevezzük. A determinizmus biztosan jó dolog : nyilván azt akarjuk, hogy ugyanazokra a kiindulási adatokra alkalmazott azonos számítási sorozat mindíg ugyanazt az eredményt szolgáltassa. Bizonyos alkalmazásoknál azonban azt szeretnénk, hogy a számítógép ne legyen kiszámítható. A játékok nyilvánvaló példák erre, de más példák is vannak.

A látszat ellenére egyáltalán nem könnyű olyan algoritmust írni, ami valóban nem-determinisztikus (vagyis ami teljesen kiszámíthatatlan eredményeket ad). Viszont vannak olyan matematikai eljárások, amik lehetővé teszik a véletlen többé kevésbé jó ***szimulációját***. Könyveket írtak azokról az eszközökről, amik a véletlent « jó minőségben » produkálják. Nyilván nem fogom most ezt a kérdést kifejteni, de az olvasót semmi sem akadályozza meg abban, hogy e tárgykörben konzultáljon a matematika tanárával.

A Python **random** modulja egész sor - különböző eloszlású - véletlenszámokat generáló függvényt kínál. Csak néhányat fogunk megvizsgálni közülük. Az on-line dokumentációban lehet a többit megnézni. A modul összes függvényét importálhatjuk a következő utasítással :

```
>>> from random import *
```

Az alábbi függvény nulla és egy értékek közé eső véletlen valós számokat hoz létre. Argumentumként a lista méretét kell megadni :

```
>>> def veletlen_lista(n):
        s = [0]*n
        for i in range(n):
            s[i] = random()
        return s
```

Először egy nullákból álló **n** elemű listát hoztunk létre, majd a nullákat véletlen számokkal helyettesítettük.

Gyakorlatok :

10.41. Írja újra a fenti **veletlen_lista()** függvényt az **append()** metódus alkalmazásával úgy, hogy a listát egy üres listából lépésről-lépésre hozza létre (azaz ne egy már meglévő lista nulla elemeit helyettesítse, ahogyan azt az előbb tettük).

10.42. Írjon egy **lista_nyomtatas()** függvényt, ami soronként írja ki egy tetszőleges méretű lista összes elemét. A lista nevét argumentumként kell megadni. Használja ezt a függvényt a **veletlen_lista()** függvénnyel generált véletlen számok kinyomtatására. Így például az **lista_nyomtatas(veletlen_lista())** utasításnak egy 8 véletlen számból álló oszlopot kell kiíratni.

Valóban véletlen számok az így létrehozott számok ? Ezt nehéz megmondani. Ha csak kevés számot generálunk, akkor semmit sem tudunk igazolni. Ha viszont a **random()** függvényt sokszor alkalmazzuk, akkor azt várjuk, hogy az előállított számok fele 0.5-nél nagyobb lesz (és a másik fele kisebb lesz).

Finomítsuk ezt az érvelést. A kisorsolt értékek mindíg a 0-1 intervallumban vannak. Osszuk fel ezt az intervallumot négy egyenlő részre : 0-0.25, 0.25-0.50, 0.5-0.75, 0.75-1.
Ha véletlenszerűen sorsolunk nagyszámú értéket, akkor azt várjuk, hogy a 4 intervallum mindegyikébe ugyanannyi fog esni. Ezt az érvelést tetszőleges számú intervallumra általánosíthatjuk, amíg azok egyenlőek.

Gyakorlat :

10.43. Egy programot kell írni, amivel a Python véletlenszám generátorának működését lehetl tesztelni a fent bemutatott elmélet alkalmazásával. Tehát a programnak :

- Meg kell kérdezni a felhasználótól a random() függvénnyel sorsolandó véletlen számok számát. Fontos, hogy a program javasoljon egy alapértelmezett számot (például 1000).
- Meg kell kérdezni a felhasználótól, hogy hány részre akarja felosztani a lehetséges értékek intervallumát (vagyis a 0-1 intervallumot). Itt is javasolni kell egy alapértelmezett értéket (például 5-öt). A felhasználó választását korlátozhatjuk 2 és a kisorsolt véletlen számok $1/10^{e}$-e közötti értékre.
- Létre kell hozni egy N számlálóból álló listát (N a választott intervallumok száma). Nyilván mindegyikük kezdő értéke nulla.
- A **random()** függvénnyel ki kell sorsolni a kért számú véletlen számot és az értékeket egy listában kell tárolni.
- Végig kell menni a véletlen értékek listáján (ciklus), és meg kell nézni, hogy a 0-1 intervallum melyik részintervallumába esik. A megfelelő számláló értékét eggyel meg kell növelni.
- Amikor ez véget ért, mindegyik számláló állapotát ki kell íratni.

Példa egy ilyen típusú program eredménylistájára :

```
A húzandó véletlen számok száma (défaut = 1000) : 100
Részintervallumok száma a 0-1 intervallumban (2 és 10 között, défaut =5) : 5
100 érték sorsolásának eredménye...
Az értékek száma az 5 részintervallumban ...
11 30 25 14 20
```

```
A húzandó véletlen számok száma (défaut = 1000) : 10000
Részintervallumok száma a 0-1 intervallumban (2 és 1000 között, défaut =5) : 5
10.000 érték sorsolásának eredménye....
Az értékek száma az 5 részintervallumban ...
1970 1972 2061 1935 2062
```

Egy ilyen fajta probléma jó megközelítése abból áll, hogy megpróbáljuk elképzelni, milyen egyszerű függvényeket tudnánk írni egyik, vagy másik részprobléma megoldására, majd ezeket egy nagyobb együttesben használni.

Például először megpróbálhatnánk definiálni egy **numeroFraction()** függvényt, mi annak a megállapítására szolgálna, hogy melyik részintervallumba esik egy kisorsolt szám. Ennek a függvénynek két argumentuma lenne (a kihúzott szám és a felhasználó által válsztott részintervallumszám) és visszatérési értékként az inkrementálandó számláló indexét adná meg (vagyis a megfelelő részintervallum sorszámát). Talán van olyan egyszerű matematikai okfejtés, ami lehetővé teszi, hogy ebből a két argumentumból meghatározzuk a részintervallum indexét. Gondoljunk az **int()** beépített függvényre, ami a decimális rész kiküszöbölésével lehetővé teszi valós számok egészekké történő alakítását.

Ha nem talál ilyet , akkor egy másik érdekes gondolat talán az lenne, hogy először létrehoznánk egy listát, ami részintervallumok végpontjait tartalmazza (például 4 részintervallum esetén 0 – 0,25 – 0,5 – 0,75 – 1). Ezeknek az értékeknek az ismerete talán egyszerűsítené a **numeroFraction()** függvény megírását.

Ha elég ideje van, akkor megcsinálhatja ennek a programnak a grafikus változatát, ami az eredményeket hisztogram (oszlopdiagram) formájában mutatja be.

Egész számok véletlen sorsolása

Amikor az olvasó saját projekteket fog fejleszteni, gyakran elő fog fordulni, hogy egy olyan függvényre lesz szüksége, amivel bizonyos határok közé eső véletlen egész számokat tud generálni. Például ha egy játékprogramot akar írni, amiben a kártyákat véletlenszerűen kell húzni (egy 52 lapos készletből), akkor biztosan hasznos lenne egy olyan függvény, ami képes lenne 1 és 52 között (az 1-et és 52-t is beleértve) egy véletlen számot sorsolni.

Erre a **random** modul **randrange()** függvénye használható.

Ez a függvény 1, 2, 3 argumentummal használható.

Egyetlen argumentummal használva : nulla és az argumentum eggyel csökkentett értéke közé eső egész számot ad visszatérési értékül. Például : **randrange(6)** egy 0 és 5 közé eső számot ad visszatérési értéknek.

Két argumentummal használva a visszatérési értéke az első argumentum és az eggyel csökkentett második argumentum értéke közé eső szám. Például : **randrange(2, 8)** egy 2 és 7 közé eső számot ad visszatérési értéknek.

Ha egy harmadik argumentumot is megadunk, akkor az azt adja meg, hogy a véletlenszerűen sorsolt egészeket egymástól a harmadik argumentummal definiált intervallum választja el. Például a **randrange(3,13,3)** a 3, 6, 9, 12 sorozat elemeit fogja visszatérési értékként adni. :

```
>>> for i in range(15):
        print random.randrange(3,13,3),

3 12 6 9 6 6 12 6 3 6 9 3 6 12 12
```

Gyakorlatok:

10.44. Írjon egy scriptet, ami véletlen szerűen húz kártyalapokat.A kihúzott kártya nevét korrekten kell, hogy megadja. A program például kiírja :

```
Nyomjon <Enter>-t egy lap húzásához :
Treff 10
Nyomjon <Enter>-t egy lap húzásához :
Káró ász
Nyomjon <Enter>-t egy lap húzásához :
Pik nyolc
Nyomjon <Enter>-t egy lap húzásához :
stb ...
```

10.3 A tuple-k

Eddig két összetett adattípust tanulmányoztunk : a stringeket, amik karakterekből állnak és a listákat, amik tetszőleges típusú adatokból állhatnak. Egy másik különbségre is emlékeznünk kell : egy stringben a karaktereket nem lehet megváltoztatni, míg egy lista elemei megváltoztathatók. Máshogy fogalmazva, a listák módosítható szekvenciák, mig a karakterláncok nem módosítható szekvenciák. Példa :

```
>>> lista=['sonka','sajt','méz','lekvár','csokoládé']
>>> liste[1:3] =['saláta']
>>> print lista
['sonka', 'saláta', 'lekvár', 'csokoládé']
>>> karakterlánc ='Roméo préfère Juliette'
>>> karakterlánc[14:] ='Brigitte'
          ***** ==> Erreur: object doesn't support slice assignment  *****
```

Megpróbáltuk módosítani a karakterlánc végét, de az nem ment. A célunk elérésére az egyetlen lehetőség, hogy létrehozunk egy új karakterláncot és oda átmásoljuk azt, amit meg akarunk változtatni :

```
>>> karakterlánc = karakterlánc[:14] +'Brigitte'
>>> print  karakterlánc
Roméo préfère Brigitte
```

A Python egy ***tuple***[45] -nek nevezett adattípust kínál, ami nagyon hasonlít egy listához, de nem lehet módosítani.
A szintaxis szempontjából a tuple elemek vesszőkkel elválasztott gyűjteménye :

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
>>> print tuple
('a', 'b', 'c', 'd', 'e')
```

Bár nem szükséges, de ajánlatos zárójelbe tenni a tuple-t, ahogyan a Python **print** utasítása maga is megteszi azt.
(Csak a kód olvashatóbbá tételéről van szó, de tudjuk, hogy ez fontos) :

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

A tuple-ken végrehajtható műveletek szintaktikailag hasonlítanak a listákon végrehajtható műveletekhez, csak a tuple-k nem módosíthatók :

```
>>> print tuple[2:4]
('c', 'd')
>>> tuple[1:3] = ('x', 'y')      ==> ***** error *****

>>> tuple = ('André',) + tuple[1:]
>>> print tuple
('André', 'b', 'c', 'd', 'e')
```

Vegyük észre, hogy egy tuple definíciójához mindíg legalább egy vesszőre van szükség (a fenti utolsó példa egy egyelemű tuple-t használ : 'André'). Az olvasó fokozatosan fográjönni a tuple-k hasznára. A listákban mindenütt előnyben részesítjük őket ott, ahol biztosak akarunk lenni abban, hogy az adatokat nem módosítjuk tévedésből a programban. Ráadásul kevésbé veszik igénybe a rendszer erőforrásait. (kevesebb memóriát foglalnak le).

45 Ez nem angol szó, egy informatikai műszó.

10.4 A szótárak

Az eddig tanulmányozott adattípusok (stringek, listák, tuple-k) mind szekvenciák, vagyis rendezett elemsorozatok voltak. Egy szekvenciában egy index (egész szám) segítségével bármelyik elemhez egyszerűen hozzáférhetünk, azzal a feltétellel, hogy ismerjük a helyét.

A szótárak egy másik összetett adattípust képeznek. Bizonyos fokig a listákra emlékeztetnek (módosíthatók mint a listák), de nem szekvenciák. A bejegyzett elemek nem megváltoztathatatlan sorrendben lesznek elrendezve. Egy kulcsnak nevezett speciális index-szel - ami lehet alfabetikus, numerikus vagy bizonyos feltételek mellett egy összetett típus - bármelyikükhöz hozzá férhetünk.

Egy szótárban, a listákhoz hasonlóan, bármilyen típusú elemeket tárolhatunk. Az elemek lehetnek : numerikus-, string-, lista-, tuple-, szótárértékek, de függvények, classok, vagy objektumok is (lásd később).

10.4.1 Szótár létrehozása

Példaként egy nyelvi szótárat hozunk létre, ami angol informatikai kifejezések magyarra fordítására szolgál. Ebben a szótárban az indexek karakterláncok lesznek.

Mivel a szótártípus egy módosítható típus, kezdhetjük egy üres szótár létrehozásával, amit majd apránként töltünk fel. Szintaktikai szempontból annak alapján ismerünk fel egy adatstruktúrát szótár típusúként, hogy az elemei **kapcsos zárójelbe** vannak zárva. Tehát a **{ }** egy üres szótárat fog jelölni :

```
>>> dico = {}
>>> dico['computer'] = 'számítógép'
>>> dico['mouse'] ='egér'
>>> dico['keyboard'] ='billentyűzet'
>>> print dico
{'computer': 'számítógép', 'keyboard': 'billentyűzet', 'mouse': 'egér'}
```

Az előző sorban megfigyelhetjük : formailag egy szótár elemek vesszővel elválasztott sorozataként jelenik meg (amit kapcsos zárójelek vesznek körül). Mindegyik elem egy objektumpárból áll : egy indexből és egy értékből, amiket kettőspont választ el.

Az indexeket ***kulcs***nak, az elemeket pedig ***kulcs-érték pár***oknak hívjuk. Megállapíthatjuk, hogy az a sorrend, amiben az elemek az utolsó sorban megjelennek, nem felel meg annak a sorrendnek, amiben megadtuk őket. Ennek semmilyen komoly következménye sincs : értékhez sose fogunk numerikus index-szel hozzáférni a szótárban. Ehelyett kulcsokat fogunk használni :

```
>>> print dico['mouse']
egér
```

Ha viszont egy szótárhoz új elemet akarunk hozzáfűzni, elég egy új kulcs-érték párt képezni, szemben a listákkal, melyek esetén egy speciális metódust (az append() -et) kell hívnunk.

10.4.2 Műveletek szótárakkal

Mostmár tudjuk, egy szótárhoz hogyan adhatunk hozzá elemeket. Elem törlésére a **del** utasítást használjuk. Példaként hozzunk létre egy másik szótárat, ami most egy gyümölcsraktár leltárát fogja tartalmazni. A kulcsok (vagy indexek) a gyümölcsnevek lesznek és az elemek értékei a raktározott gyümölcsök tömegei lesznek (ez alkalommal az értékek numerikus típusúak).

```
>>> leltar = {'alma': 430, 'banan': 312, 'narancs' : 274, 'barack' : 137}
>>> print  leltar
{'narancs': 274, 'alma': 430, 'banan': 312, 'barack': 137}
```

Ha a tulajdonos az almakészlet felszámolásáról dönt, akkor a szótárból eltávolíthatjuk ezt a bejegyzést :

```
>>> del leltar['alma']
>>> print  leltar
{'narancs': 274, 'banan': 312, 'barack': 137}
```

A **len()** függvény használható szótárral : az elemek számát adja meg.

10.4.3 A szótárak objektumok

A szótárakra specifikus metódusokat alkalmazhatunk :

A **keys()** metódus a szótárban használt kulcsok listáját adja meg :

```
>>> print dico.keys()
['computer', 'keyboard', 'mouse']
```

A **values()** metódus a szótárban tárolt értékek listáját adja meg:

```
>>> print leltar.values()
[274, 312, 137]
```

A **has_key()** metódus megadja, hogy a szótár tartalmaz-e egy meghatározott kulcsot.
A kulcsot argumentumban adjuk meg és a metódus egy « igaz » vagy « hamis » értéket ad vissza (valójában 1-et vagy 0-t), annak megfelelően, hogy a kulcs létezik, vagy sem. :

```
>>> print leltar.has_key('banan')
1
>>> if leltar.has_key('alma'):
       print 'van almank'
else:
       print 'bocsanat, de nincs almank'

bocsanat, de nincs almank
```

Az **items()** metódus a szótárból egy vele egyenértékű, tuple-kből álló listát készít :

```
>>> print leltar.items()
[('narancs', 274), ('banan', 312), ('barack', 137)]
```

A **copy()** metódussal a szótár valódi másolatát készíthetjük el. Tudnunk kell, hogy egy létező szótárnak egy új változóhoz való hozzárendelése csak egy új hivatkozást hoz létre ugyanarra az objektumra, nem pedig egy új objektumot. A listák kapcsán már beszéltem erről a jelenségről (*aliasing*). Például az alábbi utasítás (a látszat ellenére) nem definiál új szótárat :

```
>>> keszlet = leltar
>>> print keszlet
{'narancs': 274, 'banan': 312, 'barack': 137}
```

Ha módosítjuk az **leltar** -at, akkor a **keszlet** is módosul, és viszont (ez a két név valójában ugyanazt a szótárobjektumot jelöli a számítógép memóriájában) :

```
>>> del leltar['banan']
>>> print keszlet
{'narancs': 274, 'barack': 137}
```

Egy már létező szótár valódi (független) másolatának elkészítéséhez a **copy()** metódust kell használni :

```
>>> bolt = keszlet.copy()
>>> bolt['szilva'] = 561
>>> print bolt
{'narancs': 274, 'szilva': 561, 'barack': 137}
>>> print keszlet
{'narancs': 274, 'barack': 137}
>>> print  leltar
{'narancs': 274, 'barack': 137}
```

10.4.4 Szótár bejárása

A **for** ciklust felhasználhatjuk a szótárban lévő elemeket egymás után történő kezelésére, de vigyázat :

- Az iterráció során a kulcsok lesznek egymás után hozzá rendelve a munkaváltozóhoz, nem pedig az értékek.
- Nem lehet előre látni az elemekhez való hozzáférés sorrendjét (mivel a szótár nem szekvencia).

Példa :

```
>>> keszlet ={"narancs":274, "barack":137, "banan":312}
>>> for kulcs in keszlet:
...    print  kulcs

barack
banan
narancs
```

Ha az értékeket akarjuk kezelni, akkor a megfelelő kulcsok segítségével férünk hozzájuk :

```
for kulcs in keszlet:
    print kulcs, keszlet[kulcs]

barack 137
banan 312
narancs 274
```

Ez az eljárás teljesítményben kifejezve sem, és az olvashatóság szempontjából sem ideális. Inkább az előző fejezetben leírt **items()** metódus hívása a javasolt.

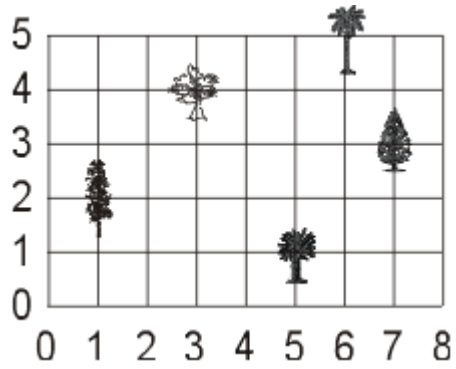
```
for kulcs, ertek in leltar.items():
    print kulcs, ertek

barack 137
banan 312
narancs 274
```

10.4.5 A kulcsok nem szükségképpen stringek

Eddig olyan szótárakat mutattam be, amiknek a kulcsai minden alkalommal *string* típusú értékek voltak. Hasonló módon, kulcsként használhatunk bármilyen nem módosítható adattípust : egészeket, valós számokat, karakterláncokat, sőt tuple-ket is.

Tételezzük fel például, hogy szeretnénk lajstromba venni egy nagy terület fáit. Erre használhatunk egy szótárat, aminek a kulcsai az egyes fák x,y koordinátáit tartalmazó tuple-k :

```
>>> fa = {}
>>> fa[(1,2)] = 'nyarfa'
>>> fa[(3,4)] = 'platan'
>>> fa[6,5] = 'palma'
>>> fa[5,1] = 'cikasz'
>>> fa[7,3] = 'fenyo'

>>> print fa
{(3, 4): 'platan', (6, 5): 'palma', (5, 1):
'cikasz', (1, 2): 'nyarfa', (7, 3): 'fenyo'}
>>> print fa[(6,5)]
palma
```



Észrevehető, hogy a harmadik sortól kezdve könnyítettem az írásmódon, kihasználva azt a tényt, hogy a tuple-ket határoló zárójelek nem kötelezőek (javaslom az óvatosságot !).

Az ilyen fajta konstrukcióknál észben kell tartani, hogy a szótár csak bizonyos adatpárokra tartalmaz elemeket. Egyébként semmit sem tartalmaz. Következésként, ha le akarjuk kérdezni a szótárat, hogy tudjuk mi hol van, illetve hol nincs, mint például a (2,1) koordinátákon, akkor egy hibaüzenetet fogunk előidézni :

```
>>> print fa[1,2]
nyarfa
>>> print fa[2,1]
                  ***** Error : KeyError: (2, 1)  *****
```

A probléma megoldására a **get()** metódust alkalmazhatjuk :

```
>>> print fa.get((1,2),'semmi')
nyarfa
>>> print fa.get((2,1),'semmi')
semmi
```

A metódus első argumentuma a keresési kulcs, a második argumentum az az érték, amit visszatérési értékként szeretnénk kapni, ha a kulcs nem létezik a szótárban.

10.4.6 A szótárak nem szekvenciák

Föntebb láttuk : a szótár elemei nincsenek egybizonyos sorrendben elrendezve. Az olyan műveleteket, mint a konkatenáció és az extrakció (folytonos elemcsoporté) itt egyszerűen nem lehet alkalmazni. Ha mégis megpróbáljuk, akkor a Python a kód végrehajtása közben hibaüzenetet küld :

```
>>> print fa[1:3]
                  ***** Error : KeyError: slice(1, 3, None) *****
```

Azt is láttuk, hogy egy új bejegyzés készítéséhez elég egy új indexet (egy új kulcsot) allokálni a szótárba. Ez nem ment a listákkal[46] :

```
>>> leltar['cseresznye'] = 987
>>> print  leltar
{'narancs': 274, 'cseresznye': 987, 'barack': 137}

>>> lista=['sonka', 'salata', 'lekvar', 'csoki']
>>> liste[4] ='szalami'
         ***** IndexError: list assignment index out of range  *****
```

Abból a tényből adódóan, hogy nem szekvenciák, a szótárak rendkívül értékesek olyan adategyüttesek kezelésénél, melyekbe gyakran kell tetszőleges sorrendben adatokat beszúrni vagy törölni. Jól helyettesítik a listákat, amikor olyan sorszámozott adategyütteseket kell kezelni, amiknek a sorszámai nem egymásra következők.

Példa :

```
>>> client = {}
>>> client[4317] = "Dupond"
>>> client[256] = "Durand"
>>> client[782] = "Schmidt"
```

stb.

Gyakorlatok :

10.45. Írjon egy scriptet, ami egy szótárral működő mini-adatbázist hoz létre, melyben a barátainak a nevét, életkorát, testmagasságát tárolja. A scriptnek két függvényt kell tartalmazni : az első a szótár feltöltésére, a második az átnézésére szolgál. A feltöltő függvényben alkalmazzon egy programhurkot a felhasználó által beírt adatok elfogadására.
A tanulók neve lesz a kulcs és az értékek tuple-kből fognak állni (életkor, testmagasság). Az életkor években (egész típusú adat), a testmagasság méterben (real típusú adat) lesz kifejezve.
Az átnézésre szolgáló függvény is tartalmaz egy programhurkot. Ebben a felhasználó valamilyen nevet ad meg, hogy visszatérési értékként megkapja a megfelelő életkor-testmagasság párt. A kérés eredményének egy formázott szövegsornak kell lenni, mint például : « Név : Jean Dhoute - kor : 15 éves - magasság : 1.74 m ». A formázáshoz használja a 130. oldalon leírt formázó karakterláncokat.

46 Emlékeztető : azokat a módszereket, amik elemek hozzáadását teszik lehetővé egy listához, a 133. oldalon beszéltük meg.

10.46. Írjon egy függvényt, ami felcseréli egy szótárban a kulcsokat és az értékeket (ami például lehetővé teszi, hogy egy angol/francia szótárat francia/angol szótárrá alakítsunk). (Feltételezzük, hogy a szótár nem tartalmaz több azonos értéket).

10.4.7 Hisztogram készítése szótár segítségével

A szótárak a hisztogramkészítés nagyon elegáns eszközei.

Tegyük fel, hogy egy hisztogramot akarunk készíteni, ami az abc betűinek egy adott szövegbeli előfordulási gyakoriságát mutatja be. A feladat megoldását leíró algoritmus rendkívül egyszerű, ha egy szótárra alapozzuk :

```
>>> szoveg ="les saucisses et saucissons secs sont dans le saloir"
>>> betuk ={}
>>> for c in szoveg:
        betuk[c] = betuk.get(c, 0) + 1

>>> print  betuk
{'t': 2, 'u': 2, 'r': 1, 's': 14, 'n': 3, 'o': 3, 'l': 3, 'i': 3, 'd': 1, 'e':
5, 'c': 3, ' ': 8, 'a': 4}
```

Létrehozzuk a **betuk** nevű üres szótárat. Ezt, az abc betűit kulcsoknak használva feltöltjük. Az egyes kulcsokkal tárolt értékek lesznek a megfelelő betűk szövegbeli előfordulási gyakoriságai. Meghatározásukhoz be kell járni a **szoveg** karakterláncot. A karaktereket kulcsként használva a **get** () metódussal kérdezzük le a szótárat, ugyanis így ki tudjuk olvasni az adott karakter eddigi előfordulásainak számát. Ha a kulcs még nem létezik a szótárban, akkor a **get()** metódusnak nulla visszatérési értéket kell adni. A megtalált előfordulási számot minden esetben megnöveljük 1-gyel és bejegyezzük a szótárba a megfelelő kulccsal.

A feladat finomításaként névsor szerint rendezve írathatjuk ki a hisztogramot. A megvalósításkor rögtön a sort() metódusra gondolunk, ez azonban csak listákra alkalmazható. Ez nem probléma ! Láttuk fentebb, hogyan alakíthatunk át egy szótárat tuple-k listájává :

```
>>> rendezett_betuk = betuk.items()
>>> rendezett_betuk.sort()
>>> print  rendezett_betuk
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3), ('n', 3),
('o', 3), ('r', 1), ('s', 14), ('t', 2), ('u', 2)]
```

Gyakorlatok :

10.47. Rendelkezésére áll valamilyen (nem túl nagy) szövegfile. Írjon egy scriptet, ami megszámolja a szövegben az abc betűinek az előfordulását (az ékezetes karakterek problémáját nem vesszük figyelembe).

10.48. Módosítsa az előbbi scriptet úgy, hogy a szövegben előforduló szavakból hozzon létre egy táblázatot. Észrevétel : a szövegekben a szavakat nemcsak betűközök, hanem írásjelek is elválaszthatják egymástól. A probléma egyszerűsítése érdekében kezdhetjük az összes nem alfabetikus karakter betűközzel való helyettesítésével, majd az eredménystringet a **split()** metódussal átalakíthatjuk szavakból álló listává.

10.49. Rendelkezésére áll valamilyen (nem túl nagy) szövegfile. Írjon egy scriptet, ami elemzi a szöveget és mindegyik szó pontos helyét (a szöveg elejétől karakterpozícióban számolva) bejegyzi egy szótárba. Ugyanannak a szónak minden előfordulását be kell jegyezni : vagyis a szótárban minden érték egy lista, ami az adott szó előfordulásainak a helyét tartalmazza.

10.4.8 Utasításfolyam vezérlés szótár segítségével

Gyakran előfordul, hogy egy változó értékétől függően másfelé kell irányítanunk egy program végrehajtását. Nyilvánvaló, hogy **if - elif - else** utasítások sorozatával kezelhető a probléma, de ez nem elegáns és elég nehézkessé válhat, ha nagyszámú lehetőséggel van dolgunk. Példa :

```
anyag = raw_input("Válasszon anyagot : ")

if anyag == 'vas':
    functionA()
elif anyag == 'fa':
    functionC()
elif anyag == 'rez':
    functionB()
elif anyag == 'ko':
    functionD()
elif   ... stb ...
```

A programozási nyelvek gyakran speciális utasításokat kínálnak az ilyen típusú problémák kezelésére, mint amilyenek a **switch** vagy **case** a **C** vagy a **Pascal** esetében. A Python egyiket sem ajánlja fel, de a problémák jó részét megoldhatjuk egy lista vagy egy szótár segítségével (a 225. oldalon adok rá részletes példát). Példa :

```
anyag = raw_input(" Válasszon anyagot  : ")

dico = {'vas':functionA,
        'fa:functionC,
        'rez':functionB,
        'ko':functionD,
         ... stb ...}
dico[anyag]()
```

A fenti két utasítást össze lehet sűríteni egybe, de külön hagyom őket a működés magyarázatához :

- Az első utasítás a **dico** szótárat definiálja. A kulcsok jelentik a különböző lehetőségeket az **anyag** változó számára és az értékek a megfelelő hívandó függvények. Jegyezzük meg, hogy csak a függvények neveiről van szó, amiket nem kell ebben az esetben zárójeleknek követni (Különben a Python mindegyik függvényt végrehajtaná a szótár létrehozásának pillanatában).
- A második utasítás az **anyag** változó segítségével tett választásunknak megfelelő függvényt hívja. A függvény nevét a kulcs segítségbével veszi ki a szótárból, majd egy zárójelpárt társít hozzá. A Python egy klasszikus függvényhívást ismer fel és végre is hajtja.

A fenti technikát úgy tökéletesíthetjük, hogy a második utasítást az alábbi változatával helyettesítjük, ami a **get**() metódus hívásával azt az esetet is figyelembe veszi, amikor a kulcs nem létezik a szótárban (így egy hosszú **elif** utasítássorozat záró **else** utasításának a megfelelőjét kapjuk meg) :

```
dico.get(anyag, masFuggveny)()
```

(Ha az **anyag** változó értéke nem felel meg a szótár egyik kulcsának sem, akkor a **masFuggveny** () függvényt fogja hívni).

Gyakorlatok :

10.50. Fejezze be a 10.45 gyakorlatot (mini adatbázis rendszer) két függvény hozzáadásával : az egyik a szótár file-ba történő kiírására, a másik a szótárnak a megfelelő file-ból történő visszaállítására szolgáljon.
A file minden sora egy szótárelemből álljon. Legyenek jól elkülönítve az adatok :
- a kulcs és az érték (vagyis a személynév egyrészről és az « életkor +testmagasság » másrészről.)
- az « életkor +testmagasság » csoportban két numerikus adat van.
Két különböző elválasztó karaktert használjon, például « @ »-ot a kulcs és az érték, illetve a « # »-ot az értéket alkotó adatok elválasztására :

```
Juliette@18#1.67
Jean-Pierre@17#1.78
Delphine@19#1.71
Anne-Marie@17#1.63

stb.
```

10.51. Tökéletesítse az előző gyakorlat scriptjét egy szótár alkalmazásával úgy, hogy a programvégrehajtást a főmenüből irányítja. A program például a következőt írja ki :

```
Válasszon:
(V)isszatölt egy már létező fileba mentett szótárat
(B)eszúr adatokat az aktuális szótárba
(Á)tnézi az aktuális szótárat
(M)enti az aktuális szótárat egy fileba
(B)efejezés :
```

A felhasználó választásának megfelelően tehát a megfelelő függvényt egy függvényszótárból kiválasztva fogjuk hívni.

11. Fejezet : Osztályok, objektumok, attributumok

Az előző fejezetekben már többször kapcsolatba kerültünk az ***objektum*** fogalmával. Tudjuk, hogy az objektum egy osztályból (egyfajta kategóriából vagy objektumtípusból) létrehozott entitás. Például a *Tkinter* könyvtárban van egy **Button()** osztály, amiből tetszőleges számú gombot hozhatunk létre egy ablakban.

Most azt vizsgáljuk meg, mi magunk hogyan definiálhatunk új objektumosztályokat. A téma viszonylag nehéz. Fokozatosan közelítjük meg. Nagyon egyszerű objektumosztályok definícióival kezdjük, amiket a későbbiekben tökéletesítünk. Számítson rá az olvasó, hogy a továbbiakban egyre összetettebb objektumokkal fog találkozni.

A reális világ objektumaihoz hasonlóan az informatikai objektumok is nagyon egyszerűek, vagy igen komplikáltak lehetnek. Különböző részekből állhatnak, amik maguk is objektumok. Az utóbbiakat más, egyszerűbb objektumok alkothatják, stb.

11.1 Az osztályok haszna

Az osztályok az objektum orientált programozás (*Object Oriented Programming vagy OOP*) fő eszközei. Ezzel a fajta programozással a komplex programokat egymással és a külvilággal kölcsönható objektumok együtteseként struktúrálhatjuk.

A programozás ilyen megközelítésének az az egyik előnye, hogy a különböző objektumokat (például különböző programozók) egymástól függetlenül, az áthatások kockázata nélkül alkothatják meg. Ez az egységbezárás (***encapsulation***) elvének a következménye. Eszerint : az objektum feladatának ellátására használt változók és az objektum belső működése valamilyen formában « be vannak zárva » az objektumba. Más objektumok és a külvilág csak jól definiált eljárásokkal férhetnek hozzájuk.

Az osztályok alkalmazása egyebek között lehetővé teszi ***a globális változók maximális mértékű elkerülését***. A globális változók használata veszélyeket rejt, különösen a nagyméretű programok esetében, mert a programtestben az ilyen változók mindíg módosíthatók vagy átdefiniálhatók. (Ennek különösen megnő a veszélye, ha több programozó dolgozik ugyanazon a programon).

Az osztályok alkalmazásából eredő másik előny az a lehetőség, hogy már meglévő objektumokból lehet új objektumokat konstruálni. Így egész, már megírt programrészeket lehet ismételten felhasználni (anélkül, hogy hozzájuk nyúlnánk !), hogy új funkcionalitást kapjunk. Ezt a ***leszármaztatás*** és a ***polimorfizmus*** teszik lehetővé.

- A leszármaztatás mechanizmusával egy « szülő » osztályból « gyermek » osztályt konstruálhatunk. A gyermek örökli a felmenője minden tulajdonságát, funkcionalitását, amikhez azt tehetünk hozzá, amit akarunk.
- A polimorfizmus teszi lehetővé, hogy különböző viselkedésmódokkal ruházzuk fel az egymásból leszármaztatott objektumokat, vagy a körülményektől függően önmagát az objektumot.

Az objektum orientált programozás opcionális a Pythonban. Sok projekt sikeresen végigvihető az alkalmazása nélkül olyan egyszerű eszközökkel, mint a függvények. Viszont tudjunk róla, hogy az osztályok praktikus és hatékony eszközök. Megértésük segíteni fog a grafikus interface-ek (*Tkinter, wxPython*) megtanulásában és hatékonyan készít elő olyan modern nyelvek alkalmazására, mint a *C++* vagy a *Java.*

11.2 Egy elemi osztály (class) definíciója

A **class** utasítással hozunk létre új objektumosztályt. Az utasítás használatát egy elemi objektumtípus - egy új adattípus - definiálásán tanuljuk meg. Az eddig használt adattípusok mind be voltak építve a nyelvbe. Most egy új, összetett adattípust definiálunk : a **Pont** típust.

A matematikai pont fogalmának felel meg. A síkban két számmal jellemzünk egy pontot (az x és y koordinátáival). Matematikai jelöléssel a zárójelbe zárt x és y koordinátáival reprezentáljuk. Például a (25,17) pontról beszélünk. A Pythonban a pont reprezentálására *float* típusú koordinátákat fogunk használni. Ezt a két értéket egyetlen entitássá vagy objektummá szeretnénk egyesíteni. Ezért egy **Pont()** osztályt definiálunk) :

```
>>> class Pont:
        "Egy matematikai pont definíciója"
```

Az osztályok definíciói bárhol lehetnek a programban, de általában a program elejére (vagy egy importálandó modulba) tesszük őket. A fenti példa valószínűleg az elképzelhető legegyszerűbb példa. Egyetlen sor elég volt az új **Pont()** objektumtípus definiálásához. Rögtön jegyezzük meg, hogy :

- A **class** utasítás az ***összetett utasítások*** egy új példája. Ne felejtsük le a sor végéről a kötelező kettőspontot és az azt követő utasításblokk behúzását. Ennek a blokknak legalább egy sort kell tartalmazni. Végtelenül leegyszerűsített példánkban ez a sor csak egy egyszerű komment. (Megállapodás szerint, ha a **class** utasítást követő első sor egy karakterlánc, akkor azt kommentnek tekinti a Python és automatikusan beágyazza az osztályok dokumentációs szerkezetébe, ami a Python integráns részét képezi. Váljon szokásunkká, hogy erre a helyre mindíg az osztályt leíró karakterláncot teszünk !).
- Emlékezzünk arra a konvencióra, hogy az osztályoknak mindíg nagybetűvel kezdődő nevet adunk. A szövegben egy másik konvenciót is betartunk : minden osztálynévhez zárójeleket kapcsolunk, ahogyan a függvénynevekkel is tesszük.

Definiáltunk egy **Pont()** osztályt, amit mostantól kezdve Pont típusú objektumok ***példányosítással (instanciation)*** történő létrehozására használhatunk. Hozzunk létre például egy új **p9**[47] objektumot :

```
>>> p9 = Pont()
```

A **p9** változó egy új **Pont()** objektum hivatkozását tartalmazza. Azt is mondhatjuk, hogy a **p9** egy új példánya a **Pont()** osztálynak.

Figyelem : utasításban hívott osztályt mindíg zárójeleknek kell követni a függvényekhez hasonlóan (még akkor is, ha semmilyen argumentumot sem adunk meg). A későbbiekben majd meglátjuk, hogy az osztályok hívhatók argumentumokkal.

Jegyezzük meg : egy osztálydefiníciónál nincs szükség zárójelekre (szemben a függvény definíciókkal) kivéve, ha azt akarjuk, hogy a definiált osztály egy másik - már létező - osztály leszármazottja legyen (ezt kicsit később fogom elmagyarázni).

Az új **p9** objektumunkon mostmár elvégezhetünk néhány elemi műveletet. Példa :

```
>>> print p9.__doc__
Egy matematikai pont definíciója
```

47 A Pythonban egyszerű értékadó utasítással hozhatunk létre egy objektumot. Más nyelvek egy speciális utasítást vezetnek be, ami gyakran a **new** utasítás, hogy megmutassák, egy mintából hozzák létre az új objektumot. Példa : **p9 = new Point()**

(A függvényekre vonatkozóan említettem (73. oldal), hogy a különböző Python objektumok dokumentációs stringjei az előredefiniált **__doc__** attribútumhoz vannak rendelve.)

```
>>> print p9
<__main__.Pont instance at 0x403e1a8c>
```

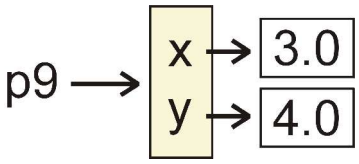
A Python-üzenet azt jelenti, hogy a **p9** a főprogram szintjén definiált **Pont()** osztály egy példánya. A memória egy jól definiált helyén helyezkedik el, aminek a címe hexadecimálisan van megadva. (Ha a tárgykörben további magyarázatra van szüksége, nézzen utána az általános informatika tantárgyban.).

11.3 Példányattribútumok vagy -változók

A létrehozott objektum egy üres kagylóhéj. A pont operátortoros minősített névmegadási rendszer[48] alkalmazásával adhatunk hozzá tartalmat :

```
>>> p9.x = 3.0
>>> p9.y = 4.0
```

Az így definiált változók a **p9** objektum ***attribútumai,*** vagy ***példányváltozói***. Be vannak ágyazva, vagy inkább ***zárva*** az objektumba. A szemközti állapotdiagram mutatja a memóriafoglalás eredményét : a **p9** változó tartalmazza a referenciát az **x** és **y** attribútumokat tartalmazó új objektum memóriabeli helyére.



Egy objektum attribútumai bármelyik kifejezésben ugyanúgy használhatók, mint a többi közönséges változó :

```
>>> print p9.x
3.0
>>> print p9.x**2 + p9.y**2
25.0
```

Az objektumba való bezártságuk miatt az attribútumok különböznek más azonos nevű változóktól. Például az **x = p9.x** utasítás jelentése : vedd ki a **p9** -cel hivatkozott objektumból az **x** attribútumot és rendeld ezt az értéket az x változóhoz ».
Nincs ütközés az **x** változó és a **p9** objektum **x** attribútuma között. A **p9** objektumnak saját névtere van, ami független az **x** változót tartalmazó főnévtértől.

48 Ez a jelölési rendszer olyan, mint amit a modulok változóinak jelölésére használunk, pl. : **math.pi** vagy **string.uppercase**. A későbbiekben lesz alkalmunk visszatérni erre. Mostmár tudjuk, hogy a modulok függvényeket, osztályokat és változókat is tartalmazhatnak. Próbáljuk ki például :

```
>>> import string
>>> print string.uppercase
>>> print string.lowercase
>>> print string.hexdigits
```

Fontos megjegyzés :

Láttuk, hogy egyszerű értékadó utasítással - mint a **p9.x = 3.0** - nagyon könnyű hozzáadni egy attribútumot egy objektumhoz. A Pythonban ezt megengedhetjük magunknak (ez a változók dinamikus értékadásának egyik következménye), de ***nem igazán javasolt ez a gyakorlat***. Ennek okát a későbbiekben fogjuk megérteni. Pusztán abból a célból járok így el, hogy a példány-attribútumokra vonatkozó magyarázatot egyszerűsítsem.

A korrekt eljárást a következő fejezetben fejtem ki.

11.4 Objektumok argumentumként történő átadása függvényhíváskor

A függvények használhatják paraméterekként az objektumokat (visszatérési értékként is megadhatnak egy objektumot). Például a következő módon definiálhatunk egy függvényt :

```
>>> def kiir_pont(p):
        print "vízszintes koord. =", p.x, "függőleges koord. =", p.y
```

A függvény **p** paraméterének egy **Pont()** típusú objektumnak kell lenni, mert a következő utasítás a **p.x** és **p.y** példányváltozókat használja. Ezért a függvény hívásakor argumentumként egy **Pont()** típusú objektumot kell megadnunk. Próbáljuk ki a **p9** objektummal :

```
>>> kiir_pont(p9)
vízszintes koord. = 3.0 függőleges koord. = 4.0
```

Gyakorlat :

(11) Írjon egy **tavolsag()** függvényt, ami kiszámolja két pont távolságát.
Ennek a függvénynek nyilván két **Pont()** objektumot kell argumentumként megadni.

11.5 Hasonlóság és egyediség

Ha két **Pont()** objektum egyenlőségéről beszélünk, akkor ez azt jelenti, hogy a két objektum ugyanazokat az adatokat (attribútumokat) tartalmazza, vagy pedig azt jelenti, hogy ugyanannak az objektumnak két hivatkozásáról beszélünk ? Tekintsük például a következő utasításokat :

```
>>> p1 = Pont()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Pont()
>>> p2.x = 3
>>> p2.y = 4
>>> print (p1 == p2)
0
```

Az utasítások létrehoznak egy **p1** és egy **p2** objektumot, amik még akkor is különbözőek maradnak, ha hasonló a tartalmuk. Az utolsó utasítás ennek a két objektumnak az egyenlőségét teszteli (dupla egyenlőségjel). Az eredmény nulla (ami azt jelenti, hogy a zárójelben lévő kifejezés hamis : tehát nem áll fenn egyenlőség).
Ezt még egy másik módon igazolhatjuk :

```
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C50F9C>
```

Az információ világos : a két változó, **p1** és **p2** különböző objektumokra hivatkoznak.

Próbáljunk most ki mást :

```
>>> p2 = p1
>>> print (p1 == p2)
1
```

A **p2 = p1** utasítással a **p1** tartalmát **p2** -höz rendeljük. Ez azt jelenti, hogy mostantól kezdve ez a két változó ugyanarra az objektumra hivatkozik. A **p1** és **p2** változók **aliasaik**[49] egymásnak.
A következő utasításban az egyenlőségteszt 1 értéket ad. Ez azt jelenti, hogy a zárójelben levő kifejezés igaz : **p1** és **p2** ugyanarra az objektumra hivatkoznak. Erről a következő módon győződhetünk még meg :

```
>>> p1.x = 7
>>> print p2.x
7
>>> print p1
<__main__.Pont instance at 00C2CBEC>
>>> print p2
<__main__.Pont instance at 00C2CBEC>
```

11.6 Objektumokból alkotott objektumok

Tételezzük most fel, hogy szeretnénk egy téglalapokat reprezentáló osztályt definiálni. Az egyszerűség kedvéért úgy tekintjük, hogy ezek a téglalapok mindíg vagy vízszintes, vagy függőleges irányításúak, soha sem állnak ferdén.

Milyen információkra van szükségünk az ilyen téglalapok definiálásához ?
Több lehetőség van. Például a téglalap középpontját tudnánk megadni (két koordináta) és a méretét (hosszúságát és szélességét). Megadhatnánk a balfelső és a jobbalsó sarkának a koordinátáit is. Vagy a balfelső sarkának a pozícióját és a téglalap méreteit. Tegyük fel, hogy ezt az utóbbi módszert választjuk.

Definiáljuk az új osztályunkat :

```
>>> class Teglalap:
        "egy téglalap osztály definíciója"
```

... és használjuk is ki rögtön egy új példány létrehozására :

```
>>> doboz = Teglalap()
>>> doboz.szeles = 50.0
>>> doboz.magas = 35.0
```

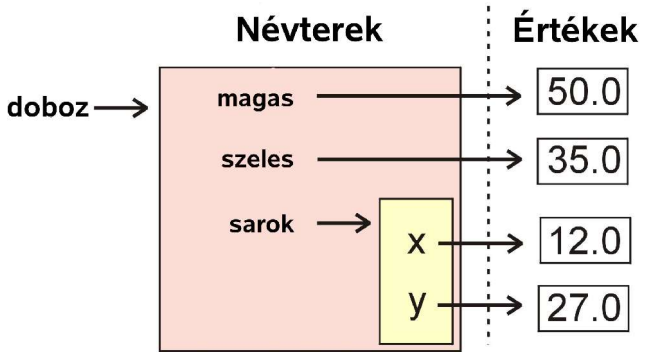
49 Az alias jelenségre vonatkozóan lásd a 139. oldalt is : Lista másolása

Így létrehozunk egy új **Teglalap()** objektumot és két attribútumot. A balfelső sarok megadásához az előbb definiált **Pont()** osztály egy objektumát fogjuk használni. Így egy objektumot hozunk létre egy másik objektum belsejében !

```
>>> doboz.sarok = Pont()
>>> doboz.sarok.x = 12.0
>>> doboz.sarok.y = 27.0
```

Egy objektum belsejében található másik objektumhoz a pont operátorral létrehozott hierarchikus névrendszer - amivel már többször találkoztunk - alkalmazásával fogunk hozzáférni. A **doboz.sarok.y** azt jelenti « menj a **doboz** változóban hivatkozott objektumba. Ebben az objektumban keresd meg a **sarok** attribútumot, majd menj az ebben az attribútumban hivatkozott objektumba. Ha megtaláltad ezt az objektumot, válaszd ki az **y** attribútumát.»

Az alábbihoz hasonló diagramok segítségével talán jobban elképzelhetők az összetett objektumok :



A « doboz» név a főnévtérben található. Egy másik névtérre hivatkozik, ami annak az objektumnak van fenntartva, amiben a « szeles », « magas » és « sarok » nevek vannak tárolva. Ezek nevek vagy egy másik névtérre hivatkoznak (a « sarok » név), vagy jól meghatározott értékekre. A Python minden modulnak, minden osztálynak, minden példánynak, minden függvénynek külön névteret foglal le. Ezeket a részekre tagolt névtereket ***robusztus*** programok (= olyan programok, amiknek a különböző komponensei között nincsenek áthatások) létrehozására használhatjuk ki.

11.7 Az objektumok mint függvények visszatérési értékei

Nous avons vu plus haut que les fonctions peuvent utiliser des objets comme paramètres. Elles peuvent également transmettre une instance comme valeur de retour. Par exemple, la fonction **trouveCentre()** ci-dessous doit être appelée avec un argument de type **Rectangle()** et elle renvoie un objet **Point()**, lequel contiendra les coordonnées du centre du rectangle.

```
>>> def trouveCentre(box):
        p = Point()
        p.x = box.coin.x + box.largeur/2.0
        p.y = box.coin.y + box.hauteur/2.0
        return p
```

Pour appeler cette fonction, vous pouvez utiliser l'objet boite comme argument :

```
>>> centre = trouveCentre(boite)
>>> print centre.x, centre.y
37.0  44.5
```

11.8 Az objektumok módosíthatók

Egy objektum tulajdonságait úgy változtathatjuk meg, hogy az attribútumaihoz új értékeket rendelünk. Például egy téglalap méretei megváltoztathatók (a pozíciója módosítása nélkül), ha a magasságának és a szélességének új értéket adunk :

```
>>> doboz.magas = doboz.magas + 20
>>> doboz.szeles = doboz.szeles – 5
```

A Pythonban ez megtehető, mert az objektumok tulajdonságai mindíg public-ok (legalábbis az aktuális 2.0 verzióban). Más nyelvek világos különbséget tesznek a public (az objektumon kívülről elérhető) és a private (csak magában az objektumban lévő algoritmusokkal elérhető) attribútumok között.

Mint már föntebb említettem (az egyszerű értékadással történő attribútum definíció kapcsán), ***egy objektum attribútumainak módosításakor nem ez a követendő gyakorlat***. Ez ugyanis ellentmond az objektum orientált programozás egyik alapvető célkitűzésének, miszerint : szigorúan el kell választani az objektum működését (ahogyan az a külvilágban deklarálva volt) és ahogyan ez a működés implementálva van az objektumban (és amit a külvilágnak nem kell ismerni).

Konkrétabban : mostantól kezdve ügyelnünk kell rá, hogy az objektumaink csak az ebből a célból lérehozott speciális **metódusokkal** legyenek módosíthatók úgy, ahogyan azt a következő fejezetben elmagyarázom.

12. Fejezet : Osztályok, metódusok, öröklés

Az előző fejezetben definiált osztályok végül is nem egyebek, mint speciális névterek, melyekben eddig csak változókat (a példány-attribútumokat) helyeztünk el.

Most el kell látnunk ezeket az osztályokat funkcionalitással. Az objektum orientált programozás alapötlete az, hogy ugyanabba az egységbe (az objektumba) kell bezárnini az adatokat (ezek a példány-attribútumok) és a kezelésükre szánt az algoritmusokat (ezek a metódusok, vagyis a bezárt függvények).

***Objektum* = [*attribútumok* + *metódusok*]**

Az, hogy egy « kapszulában » egyesítjük egy objektum tulajdonságait és azokat a függvényeket, amik lehetővé teszik, hogy hassunk ezekre a tulajdonságokra, annak a programtervezői szándéknak felel meg, hogy olyan informatikai entitásokat alkossunk, amiknek a viselkedése megközelíti a minket körülvevő világ objektumainak a viselkedését.

Vegyünk például egy « gomb » *widget*-et. Észszerűnek tűnik az a kívánság, hogy a gomb nevű informatikai objektum viselkedése hasonlítson egy igazi nyomógomb viselkedésére. Egy valóságos nyomógomb funkciója (egy elektromos áramkör nyitásának és zárásának a képessége) magába az objektumba van beépítve (ugyanúgy, mint más tulajdonságok : a mérete, színe, stb.). Ugyanígy azt kívánjuk, hogy a program-nyomógombunk különböző jellemzőit (méretét, helyét, színét, a rajta lévő szöveget), de annak definícióját is, hogy mi történik akkor, amikor különböző egérakciókat hajtunk végre a gombon, a program belsejében egy jól meghatározott entitás tartalmazza úgy, hogy ne legyen keveredés egy másik gombbal vagy más entitással.

12.1 A metódus definíciója

Mondandóm illusztrációjaként egy új Time osztályt definiálok, ami műveletek végrehajtását teszi lehetővé időpontokon, időtartamokon, stb. :

```
>>> class Time:
        "Egy idő osztály definíciója"
```

Hozzunk létre egy ilyen típusú objektumot és adjunk hozzá példányváltozókat az órák, percek és másodpercek tárolására :

```
>>> pillanat = Time()
>>> pillanat.ora = 11
>>> pillanat.perc = 34
>>> pillanat.masodperc = 25
```

Gyakorlásként írjon egy **kiir_ora()** függvényt, ami a hagyományos « óra:perc:másodperc » formában jeleníti meg a **Time()** típusú objektum tartalmát. Az imént létrehozott **pillanat** objektumra alkalmazva a függvénynek 11:34:25 -öt kell kiírni :

```
>>> print kiir_ora(pillanat)
11:34:25
```

Az olvasó függvénye valószínűleg az alábbi formájú lesz :

```
>>> def kiir_ora(t):
        print str(t.ora) + ":" + str(t.perc) + ":" + str(t.masodperc)
```

(Figyeljük meg a numerikus adatok karakterlánccá alakítására szolgáló str() függvény használatát.) Ha gyakran használja a **Time()** osztály objektumait, nagyon valószínű, hogy gyakran fogja használni ezt a függvényt is.

Valószínűleg bölcs dolog lenne a **kiir_ora()** függvényt a **Time()** osztályba bezárni, hogy automatikusan mindíg rendelkezésünkre álljon, amikor a **Time()** osztály objektumait kell kezelnünk.

Egy osztályba bezárt függvényt ***metódusnak*** nevezünk.

Már többször találkoztunk metódusokkal (így tudjuk, hogy a metódus egy objektumosztályhoz kötött függvény).

Egy metódus konkrét definíciója :

Egy metódust úgy definiálunk, mint egy függvényt. Azonban van két különbség :

- ***A metódus definícióját mindíg egy osztály definíciójának a belsejében helyezzük el*** úgy, hogy a metódust az osztályhoz kötő kapcsolat világosan kimutatható legyen.
- ***A metódus első paraméterének mindíg egy példányhivatkozásnak kell lenni.***

Elvben ennek a paraméternek bármilyen változónevet adhatnánk, de ajánlatos azt a konvenciót betartani, hogy mindíg a **self** nevet adjuk neki.

A **self** paraméter azt a példányt jelöli, amihez a metódus a definíció részét képező utasításokban hozzá lesz kapcsolva. (Ebből a tényből adódóan egy metódusnak mindíg legalább egy paramétere van, míg egy függvénydefiníció paraméter nélküli is lehet.)

Nézzük, hogy megy ez a gyakorlatban :

Ahhoz, hogy a **kiir_ora()** függvényt a **Time()** osztály metódusaként írjuk át a függvény definícióját az osztály belsejébe kell tenni és meg kell változtatni a paramétere nevét :

```
>>> class Time:
        "Új időosztály"
        def kiir_ora(self):
            print str(self.ora) + ":" + str(self.perc) \
                  + ":" + str(self.masodperc)
```

A metódus definíciója most a **class** utasítás utáni beljebb igazított utasításblokk részét képezi. Figyeljük meg jól a **self** foglalt szó használatát, ami tehát minden ebből az osztályból létrehozható példányra hivatkozik.
(Megjegyzés : A \ kód teszi lehetővé egy hosszú utasítássor folytatását a következő sorban).

A metódus kipróbálása egy objektumban

Mostmár létrehozhatjuk az új **Time()** osztályunk egy objektumát :

```
>>> most = Time()
```

Ha túl hamar próbáljuk meg használni az új metódusunkat, akkor az nem működik :

```
>>> most.kiir_ora()
AttributeError: 'Time' instance has no attribute 'ora'
```

Ez normális dolog, hiszen nem hoztunk létre példányattribútumokat. Például a következőt kellene tenni :

```
>>> most.ora = 13
>>> most.perc = 34
>>> most.masodperc = 21
>>> most.kiir_ora()
13:34:21
```

Már több ízben említettem, hogy nem tanácsos így - az objektumon kívül - példány-attribútumokat létrehozni, mert az (egyéb kellemetlenségek mellett) olyan hibákhoz vezet, mint amilyennel például az előbb találkoztunk.

Nézzük meg, hogyan csinálhatnánk jobban.

12.2 A « constructor » metódus

Elkerülhető-e az előbbi hiba ? Nem fordulna elő, ha úgy intéztük volna, hogy a **kiir_ora()** metódus mindíg ki tudjon írni valamit az újonnan létrehozott objektum előzetes manipulációja nélkül. Másként fogalmazva, ***bölcs dolog lenne, ha a példányváltozókat is előre definiálnánk az osztály belsejében*** és mindegyiküknek lenne egy alapértelmezett értéke.

Ehhez a ***constructor***-nak nevezett speciális metódust fogjuk hívni. A constructor automatikusan hajtódik végre, amikor az osztályból létrehozunk egy új objektumot. Benne elhelyezhetjük mindazt, ami a objektum automatikus inicializálásához szükséges. A Pythonban a constructort kötelező **__init__** (két « _ »karakter, az **init** szó, és újabb két « _ »karakter) -nek hívni.

Példa :

```
>>> class Time:
        "Még egy új időosztály"
        def __init__(self):
            self.ora =0
            self.perc =0
            self.masodperc =0

        def kiir_ora(self):
            print str(self.ora) + ":" + str(self.perc) \
                 + ":" + str(self.masodperc)

>>> tstart = Time()
>>> tstart.kiir_ora()
0:0:0
```

Ennek a technikának a jelentősége akkor fog világosabban megmutatkozni, ha még hozzáteszek valamit. Mint minden metódus, úgy az **__init__**() metódus is ellátható paraméterekkel. Ezek fontos szerepet fognak játszani, mert lehetővé teszik, hogy egyetlen lépésben hozzunk létre egy objektumot és inicializáljuk a példányváltozóit. Módosítsuk a példa **__init__**() metódusát a következő képpen :

```
def __init__(self, hh =0, mm =0, ss =0):
    self.ora = hh
    self.perc = mm
    self.masodperc = ss
```

Az **__init__**() metódusnak most 3 paramétere van. Mindegyiküknek van egy alapértelmezett értéke. A constructornak úgy adunk át argumentumokat, hogy az új objektum létrehozásakor az osztály neve utáni zárójelbe írjuk őket.

Az új **Time**() objektum egyidejű létrehozása és inicializálása például a következő képpen történik :

```
>>> szunet = Time(10, 15, 18)
>>> szunet.kiir_ora()
10:15:18
```

Mivel most a példányváltozóknak vannak alapértelmezett értékeik, így létrehozhatók olyan **Time** () objektumok, melyeknek egy vagy több argumentumát elhagyjuk :

```
>>> becsengetes = Time(10, 30)
>>> becsengetes.kiir_ora ()
10:30:0
```

(12) Gyakorlatok :

12.1. Definiáljon egy **Domino**() osztályt, amivel a dominójáték köveit szimuláló objektumok hozhatók létre. Az osztály constructora inicializálja a dominó A és B felén lévő pontok értékeit (az alapértelmezett értékek =0).
Definiáljon két másik metódust :
a **kiir_pontok**() metódust, ami kiírja a két oldalon levő pontokat
az **ertek**() metódust, ami a két oldalon lévő pontok összegét adja meg.

Példák az osztály alkalmazására :

```
>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.kiir_pontok()
A oldal : 2  B oldal : 6
>>> d2.kiir_pontok ()
A oldal : 4  B oldal : 3
>>> print "Összes pont :", d1.ertek() + d2.ertek ()
15
>>> lista_dominok = []
>>> for i in range(7):
        lista_dominok.append(Domino(6, i))

>>> print lista_dominok

          stb., stb.
```

12.2. Definiáljon egy **BankSzamla()** osztályt, ami lehetővé teszi a **szamla1**, **szamla2**, stb. objektumok létrehozását. Az osztály constructora két példányattribútumot inicializáljon : a nev és egyenleg attribútumokat a '**Dupont**' és **1000** alapértelmezett értékekkel.
Definiáljon három másik metódust is :
- **betesz(osszeg)** adott összeget tesz a számlára
- **kivesz(osszeg)** adott összeget vesz le a számláról
- **kiir()** kiírja a számlatulajdonos nevét és a számlája egyenlegét.

Pédák az osztály alkalmazására :

```
>>> szamla1 = BankSzamla('Duchmol', 800)
>>> szamla1.betesz(350)
>>> szamla1.kivesz(200)
>>> szamla1.kiir()
Duchmol bankszálaegyenlege 950 euro.
>>> szamla2 = BankSzamla()
>>> szamla2.betesz (25)
>>> szamla2.kiir()
Dupont bankszálaegyenlege 1025 euro.
```

12.3. Definiáljon egy **Auto()** osztályt, amivel autók viselkedését utánzó objektumok hozhatók létre. Az osztály constructora használja a következő attribútumokat a megadott alapértelmezett értékekkel :
marka = '**Ford**', **szin** = '**piros**', **sofor** = '**senki**', **sebesseg= 0**.
Egy új **Auto()** objektum létrehozásakor meg fogjuk tudni választani a márkáját és a színét, de a sebességét és a vezetője nevét nem .
Definiálja a következő metódusokat :
- **valaszt_sofort(nev) –** megadhatjuk (vagy megváltoztathatjuk) a **v**ezető nevét
- **gyorsit(gyorsulas, idotartam) –** megváltoztatja az autó sebességét. A sebességváltozás a **gyorsulás x időtartam** szorzattal egyenlő. Például ha 20 s-on keresztül az autó 1.3 m/ s^2 gyorsulású, akkor a sebességváltozás 26 m/s lesz. Negatív gyorsulást (lassulást) is elfogadunk. Ha a vezető 'senki, akkor a sebesség nem változhat.
- **kiir_mindent() –** kiírja az autó aktuális tulajdonságait, azaz a márkáját, színét, a vezetője nevét, a sebességet.

Példák az osztály használatára :

```
>>> a1 = Auto('Peugeot', 'kék')
>>> a2 = Auto(szin = 'zöld')
>>> a3 = Auto('Mercedes')
>>> a1.valaszt_sofort('Roméo')
>>> a2.valaszt_sofort('Juliette')
>>> a2.gyorsit(1.8, 12)
>>> a3.gyorsit (1.9, 11)
Ennek az autónak nincs sofőre !
>>> a2.kiir_mindent()
zöld Ford, vezeti Juliette, sebesség= 21.6 m/s.
>>> a3.kiir_mindent()
piros Mercedes, vezeti senki, sebesség = 0 m/s.
```

12.4. Definiáljon egy **Muhold()** osztályt, amiből Föld körüli pályára fellőtt mesterséges holdakat szimuláló objektumokat hozhatunk létre. Inicializálja az osztály constructora a megadott értékekkel a következö példányattribútumokat :
tomeg = 100, **sebesseg = 0**.
Egy új **Muhold()** objektum létrehozásakor megtudjuk választani a nevét, tömegét és sebességét.
A következő metódusokat kell definiálni :
- **impulsion(ero, idotartam)** meg változtatja az űrszonda sebességét. Ismételjük át fizikából a szükséges ismereteket : ***t*** ideig ***F*** erő hatása alatt álló m tömegű test Δv sebességváltozása $\Delta v = \frac{F \times t}{m}$.

Például egy 10 s-ig 600 Newton erő hatásának kitett 300 kg tömegű űrszonda sebessége 20 m/s -mal nő (vagy csökken) .
- **kiir_ sebesseg()** kiírja az űrszonda nevét és pillanatnyi sebességét.
- **energia()** kiírja az űrszonda kinetikus energiájának értékét.

Emlékeztető : a kinetikus energiát a következő képlettel számoljuk ki : $E_c = \frac{m \times v^2}{2}$

Példák az osztály használatára :

```
>>> s1 = Muhold('Zoé', tomeg =250, sebesseg =10)
>>> s1.impulsion(500, 15)
>>> s1.kiir_ sebesseg()
A(z) Zoé műhold sebessége = 40 m/s.
>>> print s1.energia()
200000
>>> s1.impulsion(500, 15)
>>> s1.kiir_ sebesseg()
A(z) Zoé műhold sebessége = 70 m/s.
>>> print s1.energia()
612500
```

12.3 Osztályok és objektumok névterei

Korábban megtanultuk (lásd a 68. oldalt), hogy egy függvény belsejében definiált változók lokális változók. Ezek a függvényen kívül található utasítások számára hozzáférhetetlenek. Ez teszi lehetővé, hogy azonos nevű változókat használhassunk egy program különböző részeiben a változók ütközésének veszélye nélkül.

Máshogy fogalmazva : minden függvénynek saját ***névtere*** van, ami független a főnévtértől.

Azt is megtanultuk, hogy a függvények belsejében lévő utasítások hozzáférhetnek a főprogram szintjén definiált változókhoz, de ***csak olvasásra*** : használhatják ezeknek a változóknak az értékeit, de nem módosíthatják őket (kivéve, ha a **global** utasítást hívják).

Tehát a névterek között létezik egyfajta hierarchia. Ugyanezt fogjuk megállapítani az osztályok és az objektumok esetében is. Valóban :

- Minden osztálynak saját névtere van. Azokat a változókat, amik ennek a névtérnek képezik a részét osztályváltozóknak vagy osztály-attribútumoknak nevezzük.
- Minden objektumnak saját névtere van. Azokat a változókat, amik ennek a névtérnek képezik a részét példányváltozóknak nevezzük.
- Az osztályok használhatják (de nem módosíthatják) a főprogram szintjén definiált változókat.
- Az objektumok használhatják (de nem módosíthatják) a osztályok szintjén és a főprogram szintjén definiált változókat.

Tekintsük például az előzőekben definiált **Time()** osztályt. A 162. oldalon ennek az osztálynak két objektumát hoztuk létre : a **szunet**-et és a **becsengetes**-t. Mindegyiket különböző, független értékekkel inicializáltuk. Mindegyik objektumban módosíthatjuk és újra kiírathatjuk ezeket az értékeket anélkül, hogy az a másik objektumra hatással lenne :

```
>>> szunet.ora = 12
>>> becsengetes.kiir_ora()
10:30:0
>>> szunet.kiir_ora()
12:15:18
```

Kódoljuk és ellenőrizzük az alábbi példát :

```
>>> class Nevterek:                                  # 1
        aa = 33                                      # 2
        def kiir(self):                              # 3
            print aa, Nevterek.aa, self.aa     # 4

>>> aa = 12                                          # 5
>>> proba = Nevterek()                               # 6
>>> proba.aa = 67                                    # 7
>>> proba.kiir()                                     # 8
12 33 67
>>> print aa, Nevterek.aa, proba.aa                  # 9
12 33 67
```

A példában ugyanazt az **aa** nevet használjuk három különböző változó definiálására : az osztály (2. sor), a főprogram (5. sor) és az objektum névterében (7. sor).
A 4. és a 9. sor bemutatja, hogy a pont-operátor alkalmazásával hogyan férhetünk hozzá ehhez a három névtérhez (egy osztály belsejéből, vagy a főprogram szintjén). Ismét figyeljük meg a **self** alkalmazását az objektum megadására.

12.4 Öröklés

A napjaink leghatékonyabb programozási technikájának tekintett objektum orientált programozásnak (*Object Oriented Programming vagy OOP*) az osztályok a fő eszközei. Ennek a programozási típusnak az egyik fő előnye az, hogy mindíg felhasználhatunk egy már meglévő osztályt egy új, néhány eltérő vagy kiegészítő funkcionalitással rendelkező osztály létrehozására. Az eljárást ***leszármaztatásnak*** nevezzük. Egy általánostól a speciális felé haladó osztály hierarchia kialakítását teszi lehetővé.

Például definiálhatunk egy - az emlős állatok jellemzőit tartalmazó - **Emlos()** osztályt. Ebből leszármaztathatjuk a **Foemlos()**, a **Ragcsalo()**, a **Ragadozo()**, stb. osztályokat. Ezek örökölni fogják az **Emlos()** osztály valamennyi jellemzőjét és hozzáadják a saját jellemzőiket.

A **Ragadozo()** osztályból leszármaztathatjuk a **Menyet()**, a **Farkas()**, a **Kutya()** stb. osztályokat. Ezek megint örökölni fogják a szülőosztályuk minden jellemzőjét, mielőtt hozzáadják a sajátjaikat.
Példa :

```
>>> class Emlos:
        jellemzo1 = "tejjel táplálja a kicsinyeit ;"

>>> class Ragadozo(Emlos):
        jellemzo2 = "a zsákmánya húsával táplálkozik ;"

>>> class Kutya(Ragadozo):
        jellemzo3 = "hangját ugatásnak hívják ;"

>>> mirza = Kutya()
>>> print mirza.jellemzo1, mirza.jellemzo2, mirza.jellemzo3
tejjel táplálja a kicsinyeit ; a zsákmánya húsával táplálkozik ;
hangját ugatásnak hívják ;
```

Látjuk, hogy a **mirza** objektum, ami a **Kutya()** osztály egy objektuma, nemcsak a **Kutya()** osztály számára definiált attribútumot örökli, hanem a szülőosztályok számára definiált attribútumokat is.

A példa bemutatja mit kell tenni, ha egy szülőosztályból akarunk származtatni egy osztályt : a **class** utasítást alkalmazzuk, amit az új osztályneve követ és ***a szülőosztály nevét zárójelek közé tesszük*** .

Jól jegyezzük meg, hogy a példában használt attribútumok osztályattribútumok (nem objektum-attribútumok). A **mirza** objektum hozzájuk férhet, de nem módosíthatja őket :

```
>>> mirza.jellemzo2 = "testet szor boritja"          # 1
>>> print mirza.jellemzo2                            # 2
testet szor boritja                                  # 3
>>> fido = Kutya()                                   # 4
>>> print fido.jellemzo2                             # 5
a zsákmánya húsával táplálkozik ;                    # 6
```

Ebben az új példában az 1. sor nem módosítja a **Ragadozo()** osztály **jellemzo2** attribútumát, ellentétben azzal, amit a 3. sor láttán gondolhatnánk. Ezt úgy igazolhatjuk, hogy létrehozzuk az új **fido** objektumot(4-6 sor) .

Ha az olvasó feldolgozta az előző fejezetet, akkor meg fogja érteni, hogy az 1. sor egy új objektumváltozót hoz létre, ami csak a **mirza** objektummal van összekapcsolva. Ettől a pillanattól kezdve két változónk van, amiknek ugyanaz - **jellemzo2** - a neve : az egyik a **mirza** objektum névterében, a másik pedig a **Ragadozo()** osztály névterében van.

Hogyan kell interpretálni a 2. és 3. sorban történteket ? Mint föntebb láttuk, a **mirza** objektum hozzáférhet a saját névterében található változókhoz, de azokhoz is, amik a szülőosztályok névtereiben vannak. Ha ezen névterek közül többen is léteznek azonos nevű változók, melyik lesz kiválasztva egy olyan utasítás végrehajtásakor, mint amilyen a 2. sorban van ?

A konfliktus föloldására a Python egy nagyon egyszerű prioritási szabályt alkalmaz. Amikor például azt kérjük tőle, hogy egy *alpha* nevű változó értékét használja, akkor a nevet a lokális névtérben (a valamilyen módon « legbelsőbb » névtérben) kezdi keresni. Ha talál egy *alpha* változót a lokális névtérben, akkor ezt fogja használni és a keresés leáll. Ha nem, akkor a Python megnézi a szülőstruktúra névterét, majd a nagyszülőstruktúra névterét, stb. egészen a főprogram névteréig.

Példánk 2. sorában tehát az objektumváltozót fogja használni. Az 5. sorban viszont csak a nagyszülőosztály szintjén található egy **jellemzo2** nevű változó. Így ez lesz kiírva.

12.5 Öröklés és polimorfizmus

Elemezzük gondosan a következő scriptet. Több, az előzőekben leírt elgondolást megvalósít, egyebek között az öröklést is.

A script megértéséhez először át kell ismételnünk néhány elemi ***kémiai*** fogalmat. A kémia órán megtanultuk, hogy az ***atomok*** bizonyos számú ***protonbó***l (pozitív elektromos töltésű részecskék), ***elektronokból*** (negatív töltésűek) és ***neutronokból*** (semlegesek) álló részecskék.

Az atom (vagy elem) típusát a protonok száma - a ***rendszám*** - határozza meg. Alapállapotában egy atom azonos számú elektront és protont tartalmaz, ezért elektromosan semleges. Változó számú neutronja is van, de ezek nem befolyásolják az elektromos töltését.

Bizonyos körülmények között az atom felvehet vagy elveszthet elektronokat. Emiatt elektromos töltésre tesz szert, ionná alakul (negatív ionná, ha egy vagy több elektront vesz fel és pozitív ionná, ha elektron(oka)t veszít). Az ion elektromos töltése a protonjai és elektronjai számának különbségével egyenlő.

A következő script « atom » és « ion » objektumokat hoz létre. Az előbb ismételtük át, hogy az ion egy módosított atom. Programunkban az « ion » objektumokat definiáló osztály tehát egy « atom » osztályból leszármaztatott osztály lesz : az utóbbi minden attribútumát és metódusát örökölni fogja és ezekhez adja a saját attribútumait és metódusait.

Az egyik ilyen hozzáadott metódus (a **kiir()** metódus) helyettesít egy « atom » osztálytól örökölt azonos nevű metódust. Az « atom » és « ion » osztályok mindegyikének van egy **kiir()** metódusa, de ezek különbözö módon működnek. Ebben az esetben ***polimorfizmusról*** beszélünk. Azt is mondhatjuk, hogy a **kiir()** metódust ***felülírtuk (overwriting).***

Nyilván tetszőleges számú atom- és ion-objektum hozható létre ebből a két osztályból. Közülük az egyiknek (az « atom osztály»-nak) tartalmaznia kell az elemek periódusos rendszerének (Mengyelejev-táblázat) egy egyszerűsített változatát úgy, hogy minden létrehozott objektumhoz hozzá tudjuk rendelni egy kémiai elem nevét és egy neutronszámot. Mivel nem kívánatos, hogy minden egyes objektumba át legyen másolva ez a táblázat, ezért egy ***osztály-attribútumban*** fogjuk elhelyezni. Így a táblázat csak egyetlen memória területen fog létezni, amihez minden ebből az osztályból létrehozott objektum hozzáférhet.

Nézzük meg, hogyan illeszkednek egymáshoz ezek a fogalmak :

```
class Atom:
    """egyszerűsített atomok, a Per.Rendszer első 10 eleme közül választva"""
    table =[None, ('hidrogén',0), ('hélium',2), ('litium',4),
            ('berilium',5), ('bór',6), ('szén',6), ('nitrogén',7),
            ('oxigén',8), ('fluor',10), ('neon',10)]

    def __init__(self, nat):
        "a rendszám meghatározza a protonok, elektronok és neutronok számát"
        self.np, self.ne = nat, nat       # nat = rendszám
        self.nn = Atom.table[nat][1]      # neutronszám a táblázatból

    def kiir(self):
        print
        print "Az elem neve :", Atom.table[self.np][0]
        print "%s proton, %s elektron, %s neutron" % \
                  (self.np, self.ne, self.nn)

class Ion(Atom):
    """az ionok atomok, amik elektronokat vettek fel vagy vesztettek"""

    def __init__(self, nat, toltes):
        "a rendszám és a töltés határozzák meg az iont"
        Atom.__init__(self, nat)
        self.ne = self.ne - toltes
        self.toltes = toltes

    def kiir(self):
        "ez a metódus a szülőosztálytól örökölt metódust helyettesíti"
        Atom.kiir (self)         #  a szülőosztály metódusát is használja !
        print "Töltött részecske. Töltés =", self.toltes

### Főprogram : ###

a1 = Atom(5)
a2 = Ion(3, 1)
a3 = Ion(8, -2)
a1.kiir()
a2.kiir()
a3.kiir()
```

A script a következőket írja ki:

```
Az elem neve : bór
5 proton, 5 electron, 6 neutron

Az elem neve : litium
3 proton, 2 electron, 4 neutron
Töltött részecske. Töltés = 1

Az elem neve : oxygén
8 proton, 10 electron, 8 neutron
Töltött részecske. Töltés = -2
```

Megállapíthatjuk, hogy a főprogramban **Atom()** objektumokat hozunk létre a rendszám megadásával (aminek 1 és 10 között kell lenni). Az **Ion()** objektumok létrehozásához viszont rendszámot és elektromos töltést (pozitív vagy negatív) kell megadni. Akár atomokról, akár ionokról van szó, ugyanaz az **kiír()** metódus - az ionok estében egy kiegészítő sorral (polimorfizmus) – íratja ki az objektumok tulajdonságait.

Magyarázatok :

Az **Atom()** osztály definíciója a **table** változó értékadásával kezdődik. Egy ebben a részben definiált változó az osztály névterének képezi részét. Ez tehát egy osztály-attribútum, amiben a Mengyelejev-féle periódusos rendszer első 10 elemére vonatkozó információk listáját teszem.

A lista a rendszámnak megfelelő indexen egy (elem neve, neutronszám) tuple-t tartalmaz. Mivel nulla rendszámú elem nem létezik, ezért a lista nulla indexű eleméhez a **None** speciális objektumot rendeltem. (A priori bármilyen más értéket hozzárendelhettem volna, mivel nem fogom használni ezt az indexet. Számomra a None objektum különösen explicitnek tűnik.).

Ezt követi a két metódus definíciója :

- Az **__init__()** constructor három ***objektum-attribútum*** létrehozására szolgál. Ezeknek az a rendeltetésük, hogy minden egyes - ebből az osztályból létrehozott - atomobjektumban külön tárolják a protonok, az elektronok és a neutronok számát. (Az objektum-attribútumok a **self** -hez kapcsolt változók) .
  Jól figyeljük meg, hogy milyen technikával kapjuk meg a neutronszámot az osztály-attribútumból : az osztály nevére a pont operátoros minősített névmegadásban hivatkozunk.
- A **kiir()** metódus egyszerre használja az objektum-attribútumokat az aktuális objektum proton-, neutron- és elektronszámának meghatározására és az osztály-attribútumot (amelyik közös minden objektumra) a megfelelő elem nevének meghatározására. Figyeljük meg a stringek formázásának technikáját is. (lásd .130. oldal).

Az **Ion()** osztály definíciója zárójelet tartalmaz. Tehát egy ***leszármaztatott osztályról*** van szó, aminek a szülőosztálya természetesen az **Atom()** osztály.

Metódusai az **Atom()** osztály metódusainak a változatai. Valószínűleg hívniuk kell ez utóbbiakat. Egy fontos megjegyzés :

Hogyan hívható egy osztály definíciójában egy másik osztályban definiált metódus ?

Ne tévesszük szem elől, hogy egy metódus mindíg ahhoz a példányhoz kapcsolódik, amit az osztályból fogunk létrehozni (a példányt a **self** reprezentálja a definícióban). Ha egy metódusnak egy másik osztályban definiált metódust kell hogy hívni, akkor ez utóbbinak meg kell adnunk annak a példánynak a hivatkozását, amelyikhez kapcsolódnia kell. Hogyan kell ezt megtenni ? Nagyon egyszerű :

Amikor egy osztály definíciójában egy másik osztályban definiált metódust akarunk hívni, első argumentumként a példány hivatkozását kell neki átadnunk.

Így például scriptünkben az **Ion()** osztály **kiír()** metódusa hívhatja az **Atom()** osztály **kiír()** metódusát : az aktuális ion-objektum adatai lesznek kiírva, mivel az ő paramétereit adtuk át a hívó utasításban :

```
Atom.kiir(self)
```

(ebben az utasításban a **self** természetesen az aktuális objektum hivatkozása).

Ugyanígy, az **Ion()** osztály constructora hívja a szülőosztálya constructorát :

```
Atom.__init__(self, nat)
```

(Számos más példát is látunk majd a későbbiekben.)

```
Atome.__init__(self, nat)
```

Osztály definíciója és használata

```
###################################
#Python program                   #
#Szerző :G. Swinnen, Liége, 2003  #
#Licenc : GPL                     #
###################################
```

Az osztály objektumok előállítására szolgáló minta. Mindegyik objektum az illető osztály egy példánya lesz.

```
class Pont:
    """matematikai pont"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

A Pont() osztály példányai nagyon egyszerű objektumok lesznek :
- *csak x és y attribútumai vannak;*
- *semmilyen metódusuk sincs.*

A self foglalt szó jelöli az osztályból létrehozott valamennyi példányt

```
class Teglalap:
    """téglalap"""
    def __init__(self,sar, ho, sze):
        self.sar = sar
        self.ho = ho
        self.sze = sze

    def kozepMeghat(self):
        xc = self.sar.x + self.ho/2
        xc = self.sar.y + self.sze/2
        return Pont(xc, yc)
```

A Teglalap () osztály példányainak három attribútuma lesz: az elsőnek ('sar') egy Pont() objektumnak kell lenni, ami a téglalap balfelső sarkának koordinátáit tárolja.

A Teglalap () osztálynak van egy metódusa, ami az őt hívó programnak egy Pont() objektumot ad visszatérési értékül..

```
class Negyzet(Teglalap):
    """négyzet = spec. téglalap"""
    def __init__(self,csucs, ol, ol):
        teglalap.__init(self,
        csucs, ol, ol)
        self.ol = ol

    def terulet(self):
        return self.ol**2
```

A Negyzet() egy leszármaztatott osztály, ami örökli a Teglalap() osztály attribútumait és metódusait . A constructorának a szülő osztálya constructorát kell hívni, argumentumként a példány (self) hivatkozását kell megadni

A Negyzet() osztálynak eggyel több metódusa van, mint a szülő osztályának.

```
###################################
##Főprogram  :                    ##

# 2 balfelső sarok koordinátái :
bfsT = Pont(40, 30)
bfsN = Pont(10, 25)

# téglalap és négyzet alakú "doboz"
dobozT = Teglalap(bfsT, 100, 50)
dobozN = Negyzet(bfsN, 40)

# a síkidomok középpontjai :
cT = dobozT.kozepMeghat()
cN = dobozN.kozepMeghat()

print "téglalap közepe: ", cT.x, cT.y
print "négyzet közepe : ", cN.x, cN.y

print "A négyzet területe :",
print dobozN.terulet()
```

Egy objektum létrehozásához az osztályát hozzárendeljük egy változóhoz.
Ezek az utasítások a Pont() osztály két objektumát hozzák létre ...

... ezek pedig két másik objektumot.
Megjegyzés : megállapodás szerint az osztályok nevét nagybetűvel kezdjük.

A kozepMeghat() metódus mindkét objektumtípuson működik, mert a Negyzet() osztály örökölte a Teglalap() osztálytól.

Viszont a terulet() metódust csak a négyzet objektumokra lehet hívni.

12.6 Osztálykönyvtárakat tartalmazó modulok

Már régóta tisztában vagyunk a Python modulok gyakorlati használával. Tudjuk, hogy osztály- és függvénykönyvtárak csoportosítására szolgálnak. Összefoglaló gyakorlatként egy új osztálymodult hozunk létre úgy, hogy az alábbi utasítássorokat egy **formes.py** nevű file-ba kódoljuk :

```
class Teglalap:
    "téglalap-osztály"
    def __init__(self, hossz =30, szelesseg =15):
        self.L = hossz
        self.l = szelesseg
        self.nev ="teglalap"

    def kerulet(self):
        return "(%s + %s) * 2 = %s" % (self.L, self.l,
                                       (self.L + self.l)*2)
    def terulet(self):
        return "%s * %s = %s" % (self.L, self.l, self.L*self.l)

    def meretek(self):
        print "Egy %s x %s -as %s" % (self.L, self.l, self.nev)
        print "területe %s" % (self.terulet(),)
        print "kerülete %s\n" % (self.kerulet(),)

class Negyzet(Teglalap):
    "Négyzet-osztály"
    def __init__(self, oldal =10):
        Teglalap.__init__(self, oldal, oldal)
        self.nev ="négyzet"

if __name__ == "__main__":
    r1 = Teglalap(15, 30)
    r1.meretek()
    c1 = Negyzet(13)
    c1.meretek()
```

Ha már egyszer tároltuk a modult, akkor kétféle módon használhatjuk : vagy úgy indítjuk el a végrehajtását, mint egy közönséges programét, vagy az osztályok használatához valamilyen scriptbe importáljuk vagy parancssorból indítjuk :

```
>>> import formes
>>> f1 = formes.Teglalap (27, 12)
>>> f1.meretek ()
Egy 27 -szer 12 -es téglalapnak
a felülete 27 * 12 = 324
és a kerülete (27 + 12) * 2 = 78

>>> f2 = formes.Negyzet (13)
>>> f2.meretek ()
Egy 13 -szor 13 -as négyzetnek
a felülete 13 * 13 = 169
és a kerülete (13 + 13) * 2 = 52
```

Látjuk, hogy a **Negyzet()** osztályt a **Teglalap()** osztályból – aminek minden jellemzőjét örökli – leszármaztatással hoztuk létre. Másként fogalmazva : a **Negyzet()** osztály a T**eglalap()** osztály gyermeke.

Megint megfigyelhetjük, hogy a **Negyzet()** osztály constructora a példány hivatkozását (azaz a **self**-et) első argumentumként megadva hívja a szülőosztálya constructorát (**Teglalap().__init__()** -et).

A modul végén az :

```
if __name__ == "__main__":
```

utasítás arra való, hogy meghatározza : a modult programként (ez esetben a következő utasításokat végre kell hajtani), vagy importált osztálykönyvtárként kell indítani. Utóbbi esetben ez a kódrész hatástalan.

Gyakorlatok :

12.5. Definiáljon egy **Kor()** osztályt. Az osztályból létrehozott objektumok különböző méretű körök lesznek. A constructoron kívül (ami a **sugar** paramétert fogja használni) definiálni kell egy **terulet()** metódust, ami a kör területét adja meg visszatérési értékként.

Definiáljon egy **Henger()** osztályt , ami a **Kor()** osztályból van származtatva. Az új osztály constructorának két paramétere legyen : a **sugar** és a **magassag**. Adjon hozzá egy **terfogat()** nevű metódust, ami a henger térfogatát adja meg visszatérési értékként.
(Ismétlés : a henger térfogata = alapkör területe x hengermagasság).

Példa az osztály alkalmazására :

```
>>> henger= Henger(5, 7)
>>> print henger.felszin()
78.54
>>> print henger.terfogat()
549.78
```

12.6. Cejezze be az előző példát egy **Kup()** osztály hozzáadásával, amit a **Henger()** osztályból származtat le és a constructora szintén tartalmazza a **sugar** és a **magassag** paramétereket. Ennek az új osztálynak a **terfogat()** lesz a saját metódusa, ami a kúp térfogatát adja meg visszatérési értékként.
(Ismétlés : a kúp térfogata = a megfelelő henger térfogata / 3).

Példa az osztály alkalmazására :

```
>>> kup= Kup(5,7)
>>> print kup.terfogat()
183.26
```

12.7. Definiáljon egy KartyaJatek() osztályt, amiből olyan kártya objektumokat lehet létrehozni, amiknek a viselkedése az igazi kártyákéra hasonlít. Az osztálynak legalább a következő három metódust kell tartalmazni :

- constructor metódus : egy 52 elemű listát kell létrehozni és kitölteni, aminek az elemei az 52 kártya mindegyikének jellemzőit tartalmazó 2 elemű tuple-kből állnak.
Mindegyik kártya esetén külön kell tárolni egy egész számot, ami megadja a kártya értékét (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, az utolsó négy érték a junga, dáma, király és az ász) és egy másik egész számot, ami a kártya színét adja meg (azaz 3, 2, 1, 0 -t a kőr, káró, a treff és a pik számára).
Egy ilyen listában a (11,2) elem a tref jungát jelöli. A kész listának [(2, 0), (3,0), (3,0), (4,0), ... (12,3), (13,3), (14,3)] típusúnak kell lenni.

- **kartya_neve()** metódus : argumentumként megadva a leíró tuple-t, ez a metódus visszatérési értékként bármelyik kártyára megadja az azt azonosító karakterláncot.
Például a :
```
print jatek.kartya_neve((14, 3))
```
utasításnak : `pik ász` -t kell kiírni

- **kever()** metódus : összekeveri a kártyákat.
A kártyákat tartalmazó lista elemeinek összekeverésére való, mindegy hány elemből áll a lista.

- **huz()** metódus : amikor ezt a metódust hívjuk, akkor egy kártyát húzunk. A színét és az értékét tartalmazó tuple -t adja meg visszatérési értékként a hívó programnak. Mindíg a lista első kártyáját húzzuk ki. Ha akkor hívjuk a metódust, amikor egyetlen kártya sem maradt a listában, akkor a speciális None objektumot kell megadnia visszatérési értékként.

Példa a KartyaJatek() osztály alkalmazására :

```
jatek = = KartyaJatek()              # egy objektum létrehozása
jatek.kever()                      # kártyák megkeverése
for n in range(53):                # 52 kártya húzása :
 c = jatek.huz()
 if c == None:                     # egy kártya sem maradt
 print 'Vége !'                    # a listában
 else:
 print jatek.kartya_neve(c)        # a kártya értéke és színe
```

12.8. Az előző gyakorlat kiegészítése : Definiálni kell két játékost, A -t és B -t. Létre kell hozni két kártyajátékot (mindegyik játékosnak egyet) és meg kell őket keverni. Utána egy ciklus segítségével 52 alkalommal egy-egy kártyát kell húzni mindkét kártyakészletből és össze kell hasonlítani az értékeiket. Ha a kettő közül ez első értéke a nagyobb, akkor az A játékosnak adunk egy pontot. Ellenkező esetben a B játékosnak adunk egy pontot. Ha egyenlő a két érték, akkor a következő húzásra térünk át. A ciklus végén össze kell hasonlítani az A és B számlálóit, hogy meghatározzuk a nyertest.

13. Fejezet : Osztályok és grafikus interface-ek

Az objektum orientált programozás különösen alkalmas grafikus interface-ű alkalmazások fejlesztésére. Az olyan osztálykönyvtáraknak, mint a *Tkinter* vagy a *wxPython* számos alap-*widgetjük* van, amiket leszármaztatással igazíthatunk igényeinkhez. Ebben a fejezetben megint a *Tkinter* könyvtárat fogjuk használni, de az előző fejezetben leírt fogalmakat alkalmazva. Arra fogok törekedni, hogy nyilvánvalóvá tegyem azokat az előnyöket, amiket az ***objektum orientáltság*** hoz a programjainkba.

13.1 « Színkódok » : egy egységbe zárt project

Egy kis projekttel kezdünk, amit a kezdő elektronika tanfolyam inspirált. Az alkalmazással gyorsan megtalálható egy meghatározott értékű ellenállás három színkódja.

Emlékeztetőül : az elektomos ellenállás funkciója az áramkorlátozás. Az ellenállások henger alakú alkatrészek, amiken körben színes csíkok vannak (általában három). Ezek a színes sávok adják meg az ellenállás numerikus értékét a következő módon :

Megállapodás szerint minden szín egy 0 és 9 közötti számnak felel meg:

fekete = 0 ; barna = 1 ; piros = 2 ; narancs = 3 ; sárga = 4 ;
zöld = 5 ; kék = 6 ; ibolya = 7 ; szürke = 8 ; fehér = 9.

Úgy fordítjuk az ellenállást, hogy a színes sávok baloldalon legyenek.

Az ellenállás ohm-okban (Ω) kifejezett értékét úgy kapjuk meg, hogy a színsávokat balról jobbra olvassuk le : a két első sáv jelenti a numerikus érték első két számjegyét; ez után annyi nullát kell tenni, amennyi a harmadik csík számértéke.

Konkrét példa :

Tegyük fel, hogy balról jobbra haladva a csíkok : sárga, ibolya és zöld színűek.
Ennek az ellenállásnak az értéke 4700000 Ω, vagy 4700 kΩ, vagy 4,7 MΩ..

Ez a rendszer nyiván csak két számjegy pontossággal teszi lehetővé az ellenállásérték meghatározását. Minden esetre ezt elégségesnek tekinthetjük a szokásos elektronikai alkalmazások többsége számára (rádió, TV, stb.).

a) Programunk feladat meghatározása :

Az alkalmazásunknak meg kell jeleníteni egy ablakot, ami egy ellenállásrajzot és egy adatbeviteli mezőt tartalmaz, amibe a felhasználó beírhat egy numerikus értéket. A « Mutat » gomb indítja el az ellenállásrajz módosítását úgy, hogy a három színsávot megfelelteti a beírt numerikus értéknek.

Kényszer : a programnak minden egész vagy valós numerikus értéket el kell fogadni 10 Ω-tól 10^{11} Ω -ig. Például a 4.78e6 értéket el kell fogadnia és korrekten kell kerekítenie, vagyis 4800000 Ω -má kell alakítania.

b) Konkrét megvalósítás

Ezt az egyszerű alkalmazást egy ***osztály*** formájában készítjük el. Ennek pillanatnyilag az egyetlen haszna az, hogy ***egy közös névteret*** hoz létre, amibe ***bezárhatjuk*** a változóinkat és függvényeinket. Ez lehetővé teszi, hogy lemondjunk a globális változókról. Valóban :

- Azokat a változókat, amikhez mindenhonnan hozzá szeretnénk férni objektum-attribútumként definiáljuk (mindegyiket a **self** segítségével kapcsoljuk az objektumokhoz).
- A függvényeket metódusként definiáljuk, tehát azok is a **self** -hez vannak kapcsolva.

A főprogram szintjén megelégszünk az így megkonstruált osztály egyetlen objektumának létrehozásával (ennek az objektumnak egyik metódusa sincs aktiválva az objektumon kívül).

```
1.   class Application:
2.       def __init__(self):
3.           """A főablak constructora"""
4.           self.root =Tk()
5.           self.root.title('Színkódok')
6.           self.drawResistor()
7.           Label(self.root,
8.                 text ="Írja be az ellenállás értékét ohm-ban :").grid(row =2)
9.           Button(self.root, text ='Mutat',
10.                 command =self.changeColours).grid(row =3, sticky = W)
11.          Button(self.root, text ='Kilép',
12.                 command =self.root.quit).grid(row =3, sticky = E)
13.          self.entry = Entry(self.root, width =14)
14.          self.entry.grid(row =3)
15.          # 0-9 értékek színkódjai :
16.          self.cc = ['black','brown','red','orange','yellow',
17.                     'green','blue','purple','grey','white']
18.          self.root.mainloop()
19.
20.      def drawResistor(self):
21.          """Vászon ellenállás modellel, amin három szines csík van"""
22.          self.can = Canvas(self.root, width=250, height =100, bg ='ivory')
23.          self.can.grid(row =1, pady =5, padx =5)
24.          self.can.create_line(10, 50, 240, 50, width =5)         # fils
25.          self.can.create_rectangle(65, 30, 185, 70, fill ='light grey', width =2)
26.          # Három szines csík (induláskor feketék) :
27.          self.line =[]              # egy listában fogjuk tárolni a három csíkot
28.          for x in range(85,150,24):
29.              self.line.append(self.can.create_rectangle(x,30,x+12,70,
30.                                                         fill='black',width=0))
31.
32.      def changeColours(self):
33.          """A beírt értéknek megfelelő három szín kiíratása"""
34.          self.v1ch = self.entry.get()       # get() metódus egy stringet ad vissza
35.          try:
36.              v = float(self.v1ch)           # átalakítás számértékké
37.          except:
38.              err =1                         # error : nem numerikus adat
39.          else:
40.              err =0
41.          if err ==1 or v < 10 or v > 1e11 :
42.              self.reportError()          # inkorrekt vagy tartományon kívüli érték
43.          else:
44.              li =[0]*3                     # a 3 kiírandó kód listája
45.              logv = int(log10(v))          # logaritmus egész része
46.              ordgr = 10**logv              # nagyságrend
47.              # az első szignifikáns számjegy előállítása :
48.              li[0] = int(v/ordgr)          # egész rész
49.              decim = v/ordgr - li[0]       # tizedes rész
50.              #  a második szignifikáns számjegy előállítása :
51.              li[1] = int(decim*10 +.5)     # +.5 a korrekt kerekítéshez
52.              # a 2 szignifikáns számjegyhez hozzáteendő nullák száma :
53.              li[2] = logv -1
54.              # A 3 szakasz színezése :
```

```
55.              for n in range(3):
56.                  self.can.itemconfigure(self.line[n], fill =self.cc[li[n]])
57.
58.      def reportError(self):
59.          self.entry.configure(bg ='red')              # mező hátterének színezése
60.          self.root.after(1000, self.emptyEntry)       # 1 sec után törölni
61.
62.      def emptyEntry(self):
63.          self.entry.configure(bg ='white')       # fehér háttér visszaállítása
64.          self.entry.delete(0, len(self.v1ch))         # karakterek törlése
65.
66.  # Főprogram :
67.  from Tkinter import *
68.  from math import log10               # 10-es alapú logaritmus
69.  f = Application()                    # alkalmazás objektum létrehozása
```

Magyarázatok :

- 1.sor : Az osztályt úgy definiáljuk, hogy nem hivatkozunk szülőosztályra (nincs zárójel). Egy új, független osztály lesz.
- 2. - 14. sorok : Az osztály constructora létrehozza a szükséges *widget*-eket : a program olvashatóságának javítása érdekében a vászon-objektum létrehozását (az ellenállásrajzzal együtt) egy külön **drawResistor()** (ellenállás rajzolás) metódusban helyeztem el. A gombok és a címke nincsennek változókban tárolva, mert nem akarok rájuk hivatkozni a program más részében.
  A *widget* -ek ablakon belüli pozícionálására a 97. oldalon leírt **grid()** metódust használjuk.
- 15. - 17. sorok : A színkódokat egy egyszerű listában tároljuk.
- 18. sor : A constructor utolsó utasítása elindítja az alkalmazást.
- 20. - 30. sorok : Az ellenállásrajz egy egyenesből és egy világosszürke téglalapból áll, amik a két kivezetést és az ellenállástestet alkotják. Három másik téglalap fogja alkotni a színes sávokat, amit a programnak a felhasználó által megadott adatnak megfelelően kell majd módosítani. Ezek a sávok kezdetben feketék; a **self.line** listában hivatkozunk rájuk.
- 32. - 53. sorok : Ez a rész tartalmazza a program funkcionalitásának a lényegét.
  A felhasználó által megadott értéket string formájában fogadja el. A 36. sorban megpróbálom ezt a stringet egy float típusú numerikus értékké átalakítani. Ha nem sikerül az átalakítás, akkor tárolom a hibát. Ha egy numerikus értékem van, akkor ellenőrzöm, hogy az az engedélyezett tartományba ($10\,\Omega$ - $10^{11}\,\Omega$) esik-e. Ha a megadott érték hibás, azt az adatbeviteli mező - ami utána üres lesz - hátterének pirosra színezésével jelzem a felhasználónak (55. - 61. sorok).
- 45. - 46. sorok : A matematikát hívjuk segítségül, hogy a numerikus értékből meghatározzuk a szám nagyságrendjét (vagyis 10 legközelebbi hatványának a kitevőjét). A logaritmusra vonatkozó részletes magyarázatokat nézze át az olvasó a matematika tankönyvben.
- 47. - 48. sorok : Ha már ismerjük a hatványkitevőt, akkor viszonylag egyszerű a kezelt szám első két számjegyének meghatározása. Példa : Tegyük fel, hogy a beírt érték 31687. Ennek a számnak a logaritmusa 4.50088, aminek az egészrésze (4) megadja a beírt érték nagyságrendjét (esetünkben ez 10^4). A szám első számjegyének meghatározásához 10^4-gyel, azaz 10000-rel kell a számot osztani és csak az eredmény egészrészét (3) kell megtartani.

- 49. - 51. sorok : Az előző szakaszban elvégzett osztás eredménye : 3.1687.
  A 49. sorban ennek a számnak a decimális részét állítjuk elő, ami példánkban : 0,1687.
  Ha ezt megszorozzuk 10-zel, akkor az eredmény egész része nem más, mint a számnak a második számjegye (példánkban ez 1).
  Könnyen hozzájuthatnánk ehhez a számjegyhez, azonban mivel ez az utolsó, azt akarjuk, hogy korrekten legyen kerekítve. Ehhez elég ha 0.5-öt hozzáadunk a 10-zel történő szorozás eredményéhez, mielőtt az egészrészt meghatározzuk. Példánkban a számolás 1,687 + 0,5 = 2,187 -et fog eredményezni, aminek az egészrésze (2) a keresett kerekített érték.
- 53. sor : A két számjegyhez hozzáadandó nullák száma a nagyságrendnek felel meg. Elég egyet levonni a logaritmusból.
- 56. sor : Az **itemconfigure()** metódust arra használjuk, hogy egy új színt rendelünk a vászonra már felrajzolt objektumhoz. Ezt a metódust alkalmazzuk minden egyes színes sáv **fill** opciójának a módosítására úgy, hogy a **self.cc** listából a három index-szel, - **li[1]**, **li[2]** és **li[3]** -mal, amik a 3 megfelelő számot tartalmazzák, - kivett szín nevét használjuk.

(13) Gyakorlatok :

13.1. Módosítsa a fenti scriptet úgy, hogy a kép háttere világoskék ('*light blue*'), az ellenállástest drapp ('*beige*') legyen, a kivezetések vékonyabbak és az értéket jelző színes sávok szélesebbek legyenek.

13.2. Módosítsa a fenti scriptet úgy, hogy a kirajzolt kép kétszer nagyobb legyen.

13.3. Módosítsa a fenti scriptet úgy, hogy 1 és 10 Ω közötti ellenállás értékeket is be lehessen írni. Ezeknél az értékeknél az első színes sávnak feketének kell maradni, a másik két sáv fogja megadni Ω -ban és tized Ω -ban az ellenállás értékét.

13.4. Módosítsa a fenti scriptet úgy, hogy a « Mutat » gombra ne legyen szükség. A módosított scriptben legyen elég egy <Enter> -t nyomni az ellenállás érték beírása után, hogy a kiírás aktiválódjon.

13.5. Módosítsa a fenti scriptet úgy, hogy a három színes sáv feketévé váljon abban az esetben, amikor a felhasználó egy elfogadhatatlan értéket ír be.

13.2 « Kisvasút » : öröklés, osztályok közötti információcsere

Az előző gyakorlatban az osztályok egyetlen jellemzőjét használtam ki : az ***egységbezárást*** (***encapsulation***). Ez olyan program írását tette lehetővé, amiben a különböző (***metódusokká*** vált) függvények mindegyike ugyanahhoz a változó-*pool*-hoz férhet hozzá : a **self**-hez kapcsoltan definiált változókhoz. Az utóbbiakat az objektum belsejében globális változóknak tekinthetjük.

Fontos, hogy megértsük : ezek nem valódi globális változók. Az objektumba vannak bezárva és nem ajánlott a kívülről történő megváltoztatásuk[50]. Másrészt ha egy osztályból több objektumot hozunk létre, akkor mindegyiküknek saját készlete van ezekből a változókból - az egyes objektumokba bezárva. Emiatt ***objektum-attribútumoknak*** nevezzük őket.

Most nagyobb sebességre kapcsolunk és készítünk egy több osztályon alapuló alkalmazást. Megvizsgáljuk, ***hogyan tudnak a különböző objektumok a metódusaik közvetítésével információt cserélni***. Azt is bemutatom, hogy - az ***öröklés*** mechanizmusát kihasználva - hogyan definiálhatjuk ***leszármaztatással*** a grafikus alkalmazásunk főosztályát egy már létező *Tkinter* osztályból.

A projekt nagyon egyszerű. Egy játékprogram megvalósításának - amire egyébként később egy példát fogok adni (lásd 229. oldalt) - első lépése lehet. Egy vásznat és két gombot tartalmazó ablakról van szó. Az első gombra kattintáskor egy vonat jelenik meg a vásznon. Amikor a második gombra kattintunk, néhány személy jelenik meg a vagonok bizonyos ablakaiban.

a) Feladat meghatározás:

Az alkalmazás két osztályt tartalmaz:

- Az **Application()** osztályt a *Tkinter* egyik alaposztályából leszármaztatással kapjuk meg : ez helyezi el a főablakot, a vásznat és a két nyomógombot.
- Egy független **Wagon()** osztály teszi lehetővé 4 hasonló vagon-objektum létrehozását a vásznon, melyek mindegyikének van egy **perso()** metódusa. Az utóbbit arra szánom, hogy a vagon három ablakának valamelyikében megjelenítsen egy személyt. Ezt a metódust a főalkalmazás különböző módon fogja hívni a különböző vagonokra, hogy kirajzoljon néhány személyt

50 Mint már korábban említettem, a Python a pont operátort alkalmazó minősített névmegadással teszi lehetővé az objektum-attribútumokhoz való hozzáférést. Más programozási nyelvek ezt tiltják vagy csak az attribútumok speciális deklarációjával teszik lehetővé (a private és public attribútumok megkülönböztetése).
Minden esetre tudjunk róla, hogy ez az eljárás nem javasolt : a jó objektum orientált programozási gyakorlat kiköti, hogy az objektumok attribútumaihoz csak speciális metódusok közvetítésével lehet hozzáférni.

b) Implementáció :

```
1.  from Tkinter import *
2.
3.  def circle(can, x, y, r):
4.      "a <can> vásznon egy <r> sugarú kör rajza <x,y> -ban"
5.      can.create_oval(x-r, y-r, x+r, y+r)
6.
7.  class Application(Tk):
8.      def __init__(self):
9.          Tk.__init__(self)        # a szülőosztály constructora
10.         self.can =Canvas(self, width =475, height =130, bg ="white")
11.         self.can.pack(side =TOP, padx =5, pady =5)
12.         Button(self, text ="Train", command =self.drawing).pack(side =LEFT)
13.         Button(self, text ="Hello", command =self.kukucs).pack(side =LEFT)
14.
15.     def drawing(self):
16.         "4 vagon létrehozása a vásznon"
17.         self.w1 = Wagon(self.can, 10, 30)
18.         self.w2 = Wagon(self.can, 130, 30)
19.         self.w3 = Wagon(self.can, 250, 30)
20.         self.w4 = Wagon(self.can, 370, 30)
21.
22.     def kukucs(self):
23.         "személyek megjelenése bizonyos ablakokban"
24.         self.w1.perso(3)        # 1. vagon, 3. ablak
25.         self.w3.perso(1)        # 3. vagon, 1. ablak
26.         self.w3.perso(2)        # 3. vagon, 2. ablak
27.         self.w4.perso(1)        # 4. vagon, 1. ablak
28.
29. class Wagon:
30.     def __init__(self, canvas_, x, y):
31.         "egy kis vagon rajza a <canvas_> vásznon <x,y> -ban"
32.         # paraméterek tárolása példány-változókban :
33.         self.canvas_, self.x, self.y = canvas_, x, y
34.         # alap téglalap : 95x60 pixel :
35.         canvas_.create_rectangle(x, y, x+95, y+60)
36.         # 3 ablak 25x40 pixel, 5 pixel távolságra :
37.         for xf in range(x+5, x+90, 30):
38.             canvas_.create_rectangle(xf, y+5, xf+25, y+40)
39.         # két 12 pixel sugarú kerék :
40.         circle(canvas_, x+18, y+73, 12)
41.         circle(canvas_, x+77, y+73, 12)
42.
43.     def perso(self, wind):
44.         "egy emberke megjelenése a <wind> ablakban"
45.         # minden egyes ablak közepe koordinátájának a kiszámítása :
46.         xf = self.x + wind*30 -12
47.         yf = self.y + 25
48.         circle(self.canvas_, xf, yf, 10)      # arc
49.         circle(self.canvas_, xf-5, yf-3, 2)   # balszem
50.         circle(self.canvas_, xf+5, yf-3, 2)   # jobbszem
51.         circle(self.canvas_, xf, yf+5, 3)     # száj
52.
53. app = Application()
54. app.mainloop()
```

Magyarázatok :

- 3. - 5. sorok : Kis körök rajzolását tervezem. Ez a függvény egyszerűsiteni fogja a dolgomat, mert lehetővé teszi, hogy a köröket a középpontjukkal és a sugarukkal definiáljam.
- 7. - 13. sorok : Az alkalmazás fő osztályát a *Tkinter* modulból importált **Tk()** ablak-osztályból leszármaztatással hozom létre.[51] Az előző fejezetben elmondottak szerint, a leszármaztatott

51 A *Tkinter* azt is megengedi, hogy az alkalmazás főablakát egy *widget-osztályból* (leggyakrabban egy **Frame()** *widgetről* van szó) leszármaztatással hozzuk létre. A *widget*-et magában foglaló ablak automatikusan lesz hozzáadva. (185. oldal).

osztály constructorának kell a szülőosztály constructorát aktiválni úgy, hogy a példány-hivatkozást első argumentumként adja meg.
A 10. - 13. sorok a vászon és a gombok elhelyezésére szolgálnak.

- 15. - 20. sorok : a 4 vagon-objektumot hozzák létre a megfelelő osztályból. Egy ciklus és egy lista segítségével elegánsabban is programozhatnám, de így hagyom, mert nem akarom feleslegesen megnehezíteni a hátralévő magyarázatokat.
  A vagon-objektumokat a vászon meghatározott helyeire akarom tenni. Ezért néhány információt meg kell adnom az objektumok constructorának : a vászon hivatkozását és a koordinátákat. Ezek a meggondolások sejtetik, hogy majd a **Wagon()** osztály definíciójakor a constructorban - ezen argumantumok fogadására - azonos számú paraméterről kell gondoskodni.
- 22. - 27. sorok : A metódus hívására a második gombra kattintáskor kerül sor. Különböző argumantumokkal hívja bizonyos vagon-objektumok **perso()** metódusát, hogy az a megadott ablakokban kirajzolja az emberkéket. Ez a néhány kódsor megmutatja, hogyan tud egy objektum egy másikkal kommunikálni az utóbbi egyik vagy másik metódusának hívásával. Az objektumokkal történő programozás alapmechanizmusáról van szó :
  Az objektumok olyan programegységek, amik a metódusaik révén cserélnek üzeneteket és hatnak egymásra.

  Ideális esetben a **kukucs()** metódusnak néhány kiegészítő utasítást kellene tartalmazni. Ezeknek az érintett vagon-objektumok létezését kellene ellenőrizni, mielőtt valamelyik metódusuk aktiválását engedélyeznénk. Az egyszerűség érdekében nem építettem be ilyen védelmet. Ennek az a következménye, hogy az első gomb előtt nem működtethető a második gomb. (Ki tudná egészíteni a metódust az ellenőrzéssel ?)
- 29. - 30. sorok : A **Wagon()** osztály egyik létező osztálynak sem leszármazottja. Mivel egy grafikus objektum-osztályról van szó, a constructorát el kell látnunk paraméterekkel, hogy fogadja a vászon hivatkozását, amire az ábrákat szánjuk, valamint az ábrák koordinátáit. Amikor az olvasó a példával kísérletezik természetesen más paramétereket is hozzáadhat : a rajz méretét, irányítását, színt, sebességet, stb.
- 31. - 51. sorok : Az utasítások nem igényelnek sok magyarázatot. A **perso()** egyik paramétere megadja, hogy melyik ablakban kell megjelenni az emberkének. Itt sem tettem óvintézkedést : a metódus például hívható a 4 vagy 5 argumentumértékkel, ami inkorrekt hatásokat idéz elő.
- 53. - 54. sorok : Az alkalmazás elindításához nem elég az **Application()** osztály egy objektumát létrehozni, mint ahogy az előző fejezet példájában tettünk. A szülőosztálytól örökölt **mainloop()** metódust is hívni kell. Viszont ez a két sor egy sorba sűríthető össze :

```
Application().mainloop()
```

Gyakorlat :

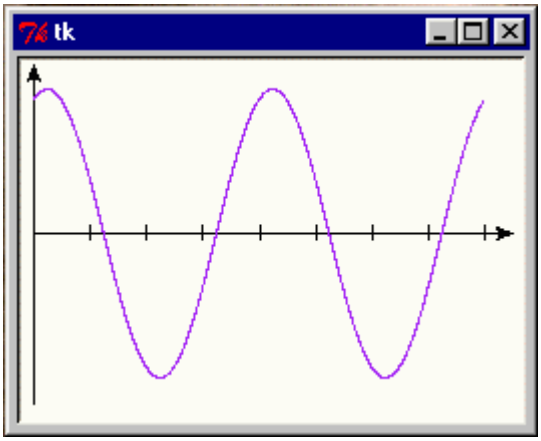
13.6. Tökéletesítse a fenti scriptet úgy, hogy a **Wagon()** osztály constructorához hozzáad egy **colour** paramétert, ami a vagon fülkéjének a színét fogja meghatározni. Kezdetben az ablakok legyenek feketék és a kerekek szürkék (adjon egy **colour** paramétert a **circle()** függvényhez is) .
Adjon a **Wagon()** osztályhoz egy **light()** metódust, ami a kezdetben fekete 3 ablak színét sárgára változtatja. Ezzel szimuláljuk a belső világítás bekapcsolását.
Adjon a főablakhoz egy világítás kapcsoló gombot. Használja ki a **circle()** függvény tökéletesítését arra, hogy az emberkék arcát rózsaszínre (*pink*), szemüket és szájukat feketére színezi, és különböző színű vagon-objektumokat hoz létre.

13.3 « OscilloGraphe » : egy testre szabott widget

A következő projekt egy kicsit messzebbre vezet. Egy új *widget*-osztályt konstruálunk, ami ugyanúgy beépíthető a későbbi projektjeinkbe, mint bármely standard *widget*. Az előző gyakorlat főosztályához hasonlóan, ezt az osztályt is ***leszármaztatással*** hozzuk létre egy már létező *Tkinter* osztályból.

Az alkalmazást a fizika óra inspirálta. Emlékeztetőül :

A harmónikus rezgőmozgást *úgy definiáljuk, mint az egyenletes körmozgás vetületét egy egyenesre. Egy ilyen típusú mozgást végző tömegpont pozícióját egy központi helyzethez szoktuk viszonyítani és* ***kitérésnek*** *nevezzük. A tömegpont kitérését időben leíró egyenlet mindíg* $e = A\sin(2\pi f t + \varphi)$ *alakú, ahol* ***e*** *a tömegpont kitérése a* ***t*** *időpillanatban. Az* ***A***, ***f*** *és* **φ** *állandók a rezgőmozgás amplitúdóját, frekvenciáját és fázisát jelölik.*



A projekt célja az, hogy egy olyan egyszerű eszközt adjon, amivel képszerűen megjeleníthetőek ezek a fogalmak, azaz egy grafikus kitérés—idő görbét rajzoló rendszert készítünk. A felhasználó szabadon választhatja meg az **A**, **f** és **φ** paraméterek értékeit és megfigyelheti a nekik megfelelő görbéket.

Az első *widget* a kiíratással fog foglalkozni. Utána más, az **A**, **f** és **φ** paraméterek beírását megkönnyítő *widget*-eket fogunk készíteni.

Írjuk be következő scriptet és mentsük egy **oscillo.py** nevű fileba. Így egy egyetlen osztályt tartalmazó valódi modult készítünk (később más osztályokat is hozzátehetünk a modulhoz, ha úgy akarjuk)

```
1.   from Tkinter import *
2.   from math import sin, pi
3.
4.   class OscilloGraphe(Canvas):
5.       "kitérés/idő görbe rajzolására szolgáló specializált vászon"
6.       def __init__(self, boss =None, width_=200, height_=150):
7.           "A grafika constructora : tengelyek és vízszintes skála."
8.           # a szülő widget elkészítése :
9.           Canvas.__init__(self)                              # a szülőosztály
10.          self.configure(width=width_, height=height_)       # constructorának hívása
11.          self.width_, self.height_ = width_, height_        # tárolás
12.          # tengelyek megrajzolása :
13.          self.create_line(10, height_/2, width_, height_/2, arrow=LAST)  # X tengely
14.          self.create_line(10, height_-5, 10, 5, arrow=LAST)              # Y tengely
15.          # 8 osztású skála rajzolása :
16.          step = (width_-25)/8.          # vízszintes skála intervallumai
17.          for t in range(1, 9):
18.              stx = 10 + t*step        # +10, hogy az origótól eljöjjünk
19.              self.create_line(stx, height_/2-4, stx, height_/2+4)
20.
21.      def drawCurve(self, freq=1, phase=0, ampl=10, colo='red'):
22.          "1 sec időtartamra eső kitérés/idő görbe rajzolása"
23.          curve =[]                          # koordináták listája
24.          step = (self.width_-25)/1000.      # az X-skála 1 sec-nak felel meg
25.          for t in range(0,1001,5):          # amit 1000 ms-ra osztunk fel
26.              e = ampl*sin(2*pi*freq*t/1000 - phase)
27.              x = 10 + t*step
28.              y = self.height_/2 - e*self.height_/25
29.              curve.append((x,y))
30.          n = self.create_line(curve, fill=colo, smooth=1)
31.          return n                           # n = a rajz sorszáma
32.
33.  #### Kód az osztály tesztelésére : ####
34.
35.  if __name__ == '__main__':
36.      root = Tk()
37.      gra = OscilloGraphe(root, 250, 180)
38.      gra.pack()
39.      gra.configure(bg ='ivory', bd =2, relief=SUNKEN)
40.      gra.drawCurve(2, 1.2, 10, 'purple')
41.      root.mainloop()
```

A script törzsét a 35. - 41. sorok alkotják. A 174. oldalon elmondottak szerint, az **if __name__ == '__main__':** utasítás utáni kódsorok csak akkor hajtódnak végre, ha a scriptet főalkalmazásként indítjuk el. Modulként importálva nem hajtódnak végre ezek a sorok.

Ez az érdekes mechanizmus teszi lehetővé, hogy a modulok belsejébe tesztutasításokat építhessünk be még akkor is, ha a modulokat más scriptekbe történő importálásra szánjuk.

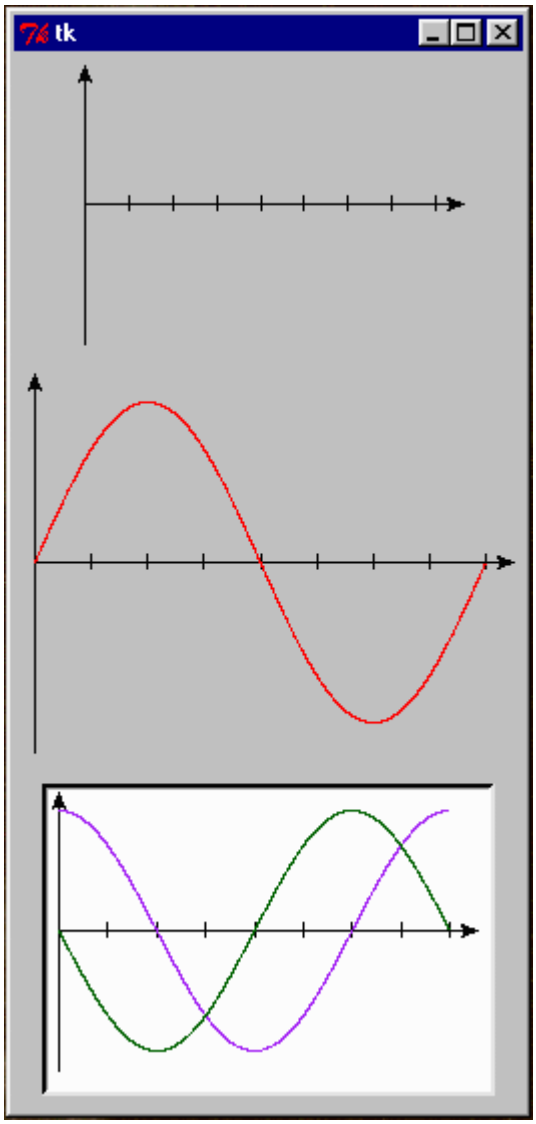
Indítsuk el a script végrehajtását a szokásos módon. Egy olyan display-képet kell kapnunk, mint amilyen az előző oldalon van.

Kísérletezés :

A script lényeges sorait egy kicsit később fogom megmagyarázni. Először kísérletezzünk egy kicsit az osztályunkkal.nstruire.

Nyissunk egy terminálablakot (« *Python shell* »), és a parancssorba írjuk be az alábbi utasításokat :

```
>>> from oscillo import *
>>> g1 = OscilloGraphe()
>>> g1.pack()
```



Az **oscillo** modul osztályainak importja után létrehozzuk az **OscilloGraphe()** osztály első objektumát **g1** -et.

Mivel semmilyen argumentumot sem adunk meg, ezért az objektum méretei az osztály constructorában definiált alapértelmezett méretek lesznek. Vegyük észre, nem kínlódtunk master ablak definiálásával, hogy utána abban helyezzük el a widgetünket. A *Tkinter* megbocsátja nekünk ezt a feledékenységet és automatikusan létrehoz egyet !

```
>>> g2 = OscilloGraphe(height_=200, width_=250)
>>> g2.pack()
>>> g2.drawCurve()
```

Ezekkel az utasításokkal az **OscilloGraphe()** osztály egy másik widgetjét hozzuk létre. Ez alkalommal megadjuk a méreteit (a magasságát és a szélességét, a sorrend tetszőleges).

Utána a widgethez kapcsolt **drawCurve()** metódust (görberajzolás) aktiváljuk. A megjelenő sinus-görbe az alapértelmezett **A**, **f** és **φ** paramétereknek felel meg, mivel semmilyen argumentumot sem adunk meg.

```
>>> g3 = OscilloGraphe(width_=220)
>>> g3.configure(bg='white', bd=3, relief=SUNKEN)
>>> g3.pack(padx=5,pady=5)
>>> g3.drawCurve(phase=1.57, colo='purple')
>>> g3.drawCurve(phase=3.14, colo='dark green')
```

A harmadik widget konfigurációjának megértéséhez vissza kell arra emlékeznünk, hogy az **OscilloGraphe()** osztályt a **Canvas()** osztályból leszármaztatással hoztuk létre. Ezért az előbbi örökli az utóbbi minden tulajdonságát. Ez lehetővé teszi, hogy ugyanazokat az argumentumokat használva, mint amik egy vászon konfigurálása során állnak rendelkezésünkre mi válasszuk meg a hátteret, a keretet, stb.

Ezután a fázis és a szín argumentumokat megadva két görbét jelenítünk meg a **drawCurve()** (görberajzolás) metódus kétszeri hívásával.

Gyakorlat :

13.7. Hozzon létre egy 400 x 300 méretű, sárga hátterű negyedik widgetet és rajzoljon ki több, különböző frekvenciájú és amplitúdójú görbét.

Elemezzük az **OscilloGraphe()** osztály szerkezetét ! Az osztályt az **oscillo.py** modulba mentettük (lásd a 184. oldalt).

a) Feladat meghatározás :

Egy olyan új *widget*-osztályt akarunk definiálni, ami képes a különböző harmónikus rezgőmozgásoknak megfelelő kitérés–idő grafikonok automatikus megjelenítésére.

A *widget*-nek létrehozásakor méretezhetőnek kell lenni. Két nyílhegyben végződő X, Y Descartes koordináta-tengelyt kell megjelenítsen. Az **X**-tengely reprezentálja 1 sec-on belül az idő múlását és 8 részre legyen osztva.

Egy **drawCurve()** (görberajzolás) metódus legyen a *widget*-hez kapcsolva. Ez a harmónikus rezgőmozgás kitérés–idő grafikonját rajzolja ki. Mi adjuk meg a frekvenciát (0.25 és 10 Hz között), a fázist (0 et 2π radián között) és az amplitúdót (1 és 10 önkényes skálaosztás között).

b) Implementáció :

- 4. sor : Az **OscilloGraphe()** osztályt a **Canvas()** osztályból leszármaztatással hozzuk létre.
  Az utóbbi osztály minden tulajdonságát örökli : az új osztály objektumait a **Canvas()** osztály már rendelkezésre álló opcióinak felhasználásával konfigurálhatjuk.
- 6. sor : A constructor metódus 3 paramétert használ. Ezek azért opcionálisak, mert mindegyiküknek van alapértelmezett értéke. A **boss** paraméter csak egy esetleges master ablak hivatkozásának a fogadására szolgál (lásd a következő példákat). A **width_** és **height_** (szélesség és magasság) paraméterek arra szolgálnak, hogy az objektum létrehozásának pillanatában a szülő vászon **width** és **height** opcióihoz rendeljünk értékeket.
- 9. és 10. sor : Az első művelet, amit egy leszármaztatott osztály constructorának végre kell hajtani : aktiválnia kell a szülőosztálya constructorát. Csak akkor örökölhető a szülöosztály minden viselkedése, ha valóban implementáltuk ezeket a viselkedéseket.
  Tehát a 9. sorban aktiváljuk a **Canvas()** osztály constructorát és a 10. sorban az opciói közül kettőt beállítunk. Vegyük észre, hogy ezt a két sort összesűríthetnénk egyetlen sorba:

```
Canvas.__init__(self, width=width_, height=height_)
```

  Emlékeztető : a 170. oldalon elmagyaráztam, az aktuális objektum hivatkozását (self) első argumentumként kell megadnunk a szülőosztály constructorának.
- 11. sor : A **width_** és **height_** (szélesség és magasság) paramétereket objektum-változókban kell tárolnunk, mert hozzájuk kell tudnunk férni a **drawCurve()** (görberajzolás) metódusban.
- 13. és 14. sor : Az X- és Y-tengely kirajzolásához a **width_** és **height_** (szélesség és magasság) paramétereket használjuk, így ezeket a görbéket automatikusan méretezzük. Az **arrow=LAST** opció minden vonal végén nyílhegyet jelenít meg.
- 16. - 19. sorok : A vízszintes skála megrajzolását a rendelkezésre álló hossz 25 pixellel való csökkentésével kezdjük úgy, hogy a skála két végén helyet hagyunk. Utána 8 intervallumra osztjuk fel, amiket 8 kis függőleges szakasz reprezentál.
- 21. sor : A **drawCurve()** (görberajzolás) metódust négy argumentummal lehet hívni.
  Ezek mindegyikét elhagyhatjuk, mert mindegyiknek van alapértelmezett értéke. Arra is lehetőség van, hogy az argumentumokat tetszőleges sorrendben adjuk meg, ahogyan azt a 79. oldalon már magyaráztam.

- 23. - 31. sorok : A görbe megrajzolásához a **t** változó egymás után vesz fel minden értéket 0-tól 1000-ig és minden alkalommal kiszámoljuk a megfelelő **e** kitérésértéket az elméleti összefüggés segítségével (26. sor). Az így megtalált **t** és **e** értékpárokat skálázzuk és a 27.-28. sorokban x, y koordinátákká alakítjuk, majd a **curve** (görbe) listában összegyűjtjük öket.
- 30. - 31. sorok : a **create_line()** metódus egyetlen műveletben megrajzolja a megfelelő görbét és visszatérési értékként a vászonon így létrehozott objektum sorszámát adja meg (ez a sorszám fogja lehetővé tenni, hogy utána a görbéhez hozzáférjünk : például azért, hogy töröljük). A **smooth =1** opció simítással javítja a végső görbealakot.

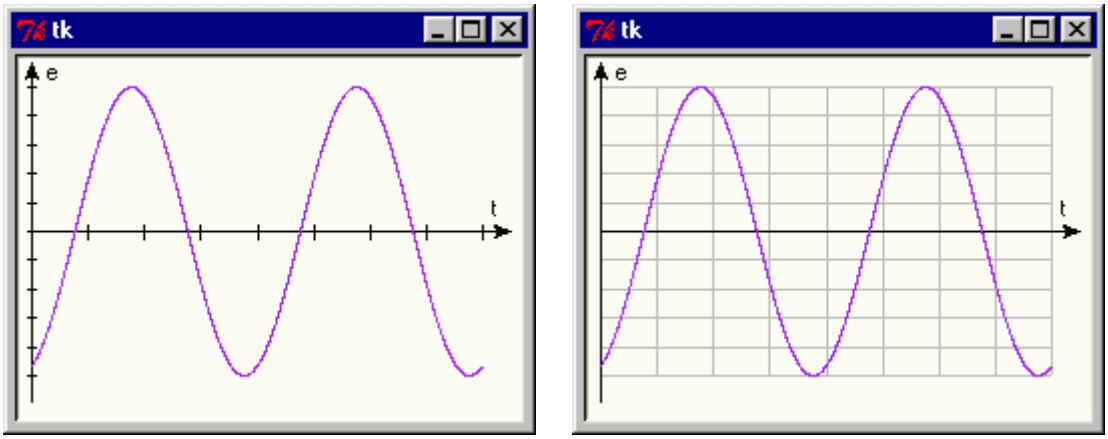
Gyakorlarok:

13.8. Módosítsa úgy a scriptet, hogy a függőleges referencia-tengelyen is legyen az origó két oldalán 5-osztásos skála.

13.9. Úgy, mint a **Canvas()** osztály *widget*-jei, - amiből az olvasó *widget*-je származik - az olvasó *widget* -je is beépíthet szöveges jelzéseket. Ehhez a **create_text()** metódust kell használni. Ez a metódus legkevesebb 3 argumentumot vár : annak a helynek az x és y koordinátáit, ahol a szövegünket meg akarjuk jeleníteni és természetesen magát a szöveget. Más argumentumokat opciók formájában lehet megadni, például a font és a karakterméret definiálásához. Hogy lássuk a működését, adjuk ideiglenesen a következő sort az **OscilloGraphe()** osztály constructorához, majd indítsuk újra a scriptet :

```
self.create_text(130, 30, text = "Proba", anchor =CENTER)
```

Használjuk arra ezt a metódust, hogy a *widget*-ben a referencia-tengelyek végéhez a következő jelzéseket kapcsoljuk : **e** -t (kitérés) a függőleges tengelyhez, és **t** -t (idő) a vízszintes tengelyhez. Az eredmény a baloldali ábrára hasonlíthat :



13.10. Vz olvasó befejezheti a *widget* -jét egy referenciarács megjelenítésével, amit a tengelyek mentén elhelyezett szakaszok helyett rajzoltathat ki. A rács vonalait szürkére színezheti (**fill = 'grey' opció**), hogy ne legyenek annyira feltűnőek, úgy mint a jobboldali ábrán.

13.11. Egészítse ki a *widget*-jét skálaszámozással.

13.4 « Kurzorok » : egy kompozit widget

Az előző gyakorlatban egy új widget-típust hoztunk létre, amit az **oscillo.py** modulba mentettünk el. Gondosan őrizzük meg a modult, mert hamarosan beépítjük egy összetettebb projektbe.

Most egy másik, interaktívabb widget-et konstruálunk. Egy vezérlő panelről van szó, ami három csúszkát és egy checkboxot tartalmaz. Mint az előző widget-et, ezt is egy későbbi összetett alkalmazásban akarjuk újra felhasználni.

13.4.1 A « Scale » widget bemutatása

Kezdjük először egy eddig még nem használt alap *widget* felfedezésével.
A **Scale** *widget* egy skála előtt csúszó cursor-ként jelenik meg. A felhasználó gyorsan kiválaszthat vele egy tetszőleges paraméterértéket.

Az alábbi kis script megmutatja, hogy hogyan paraméterezzük és használjuk egy ablakban :

```
from Tkinter import *

def updateLabel(x):
    lab.configure(text='Aktuális érték = ' + str(x))

root = Tk()
Scale(root, length=250, orient=HORIZONTAL, label ='Beállítás :',
      troughcolor ='dark grey', sliderlength =20,
      showvalue =0, from_=-25, to=125, tickinterval =25,
      command=updateLabel).pack()
lab = Label(root)
lab.pack()

root.mainloop()
```

A sorokat nem kell magyarázni. Tetszőleges méretű (**length** opció), fekvő (mint példánkban) vagy álló (**orient = VERTICAL** opció) **Scale** widget-eket létrehozhatunk.
A **from_** (figyelem : ne felejtsük el a '_' karaktert !) és **to** opciók definiálják a szabályozási tartományt. A skálaértékek közötti intervallumot a **tickinterval** opcióban definiáljuk, stb.

A cursor elmozdulásakor minden alkalommal automatikusan sor kerül a **command** opcióban meghatározott függvény hívására, és a cursor skálához viszonyított aktuális pozíciója lesz argumentumként átadva. Ez az érték bármilyen művelethez nagyon egyszerűen felhasználható. Nézzük meg a példabeli **updateLabel()** függvény **x** paraméterét.

A **Scale** *widget* egy nagyon intuitív és vonzó interface, ami különböző beállításokat kínál a programunk felhasználóinak. Több példányban fogjuk beépíteni egy új widget-osztályba : egy vezérlőpanelbe, amivel egy rezgőmozgás frekvenciáját, fázisát és amplitúdóját állítjuk be, majd az előző oldalakon készített **oscilloGraphe** widget segítségével ki fogunk íratni a rezgőmozgás kitérés–idő grafikonját.

13.4.2 Háromcursoros vezérlőpanel készítése

Az előző scripthez hasonlóan a következő scriptet is egy modulba mentjük (a **curseurs.py** modulba). Az így elmentett osztályokat egy komplex alkalmazásban fogjuk (importálással) újra felhasználni, amit a későbbiekben írok le [52]. Felhívom a figyelmet arra, hogy az alábbi kód többféle módon rövidíthető. (Erre még vissza fogunk térni.) Nincs optimalizálva, mivel ehhez egy kiegészítő fogalomra (a *lambda* kifejezésekre) lenne szükségünk, amit most inkább mellőzök.

Azt már tudjuk, hogy a script végére írt kódsorok a script működésének tesztelését teszik lehetővé. Az alábbira hasonlító ablakot kellene kapnunk :


```
1.   from Tkinter import *
2.   from math import pi
3.
4.   class ChoiceVibra(Frame):
5.       """Cursor-ok egy rezgés frekvenciájának, fázisának és amplitúdójának kiv.hoz"""
6.       def __init__(self, boss =None, colo ='red'):
7.           Frame.__init__(self)        # a szülöosztály constructora
8.           # Néhány példány-attribútum inicializálása :
9.           self.freq, self.phase, self.ampl, self.colo = 0, 0, 0, colo
10.          # A checkbox állapotváltozója :
11.          self.chk = IntVar()                # Tkinter 'objektum-változó'
12.          Checkbutton(self, text='Rajzol', variable=self.chk,
13.                      fg = self.colo, command = self.setCurve).pack(side=LEFT)
14.          # A 3 cursor-widget definíciója :
15.          Scale(self, length=150, orient=HORIZONTAL, sliderlength =25,
16.                label ='Frekvencia (Hz) :', from_=1., to=9., tickinterval =2,
17.                resolution =0.25,
18.                showvalue =0, command = self.setFrequency).pack(side=LEFT)
19.          Scale(self, length=150, orient=HORIZONTAL, sliderlength =15,
20.                label ='Fázis (fok) :', from_=-180, to=180, tickinterval =90,
21.                showvalue =0, command = self.setPhase).pack(side=LEFT)
22.          Scale(self, length=150, orient=HORIZONTAL, sliderlength =25,
23.                label ='Amplitúdó :', from_=1, to=9, tickinterval =2,
24.                showvalue =0, command = self.setAmplitude).pack(side=LEFT)
25.
26.      def setCurve(self):
27.          self.event_generate('<Control-Z>')
28.
29.      def setFrequency(self, f):
30.          self.freq = float(f)
31.          self.event_generate('<Control-Z>')
32.
33.      def setPhase(self, p):
34.          pp =float(p)
35.          self.phase = pp*2*pi/360        # fok -> radián átalakítás
36.          self.event_generate('<Control-Z>')
37.
38.      def setAmplitude(self, a):
39.          self.ampl = float(a)
40.          self.event_generate('<Control-Z>')
41.
```

52 Nyilvánvalóan több osztályt is menthetnénk ugyanabba a modulba.

```
42.  #### Kód az osztály teszteléséhez : ###
43.
44.  if __name__ == '__main__':
45.      def showAll(event=None):
46.          lab.configure(text = '%s - %s - %s - %s' %
47.                                (fra.chk.get(), fra.freq, fra.phase, fra.ampl))
48.      root = Tk()
49.      fra = ChoiceVibra(root,'navy')
50.      fra.pack(side =TOP)
51.      lab = Label(root, text ='test')
52.      lab.pack()
53.      root.bind('<Control-Z>', showAll)
54.      root.mainloop()
```

Ezen a vezérlő panelen a felhasználó könnyen be tudja állítani a megadott paraméterek (frekvencia, fázis, amplitúdó) értékeit, amik majd az előzőekben konstruált **OscilloGraphe()** class egy widgetjében a kitérés–idő grafikon kirajzolásának vezérlésére szolgálhatnak. Ezt majd az összetett alkalmazásban be fogom mutatni.

Magyarázatok :

- 6. sor : A « constructor » metódus a **colo** opcionális paramétert használja. Ez a paraméter lehetővé fogja tenni, hogy a widget kontrolja alatt lévő grafikonnak színt válasszunk. A **boss** paraméter szolgál egy esetleges master ablak (lásd később) referenciájának a fogadására.
- 7. sor : A szülő osztály constructorának aktiválása (azért, hogy a gyermek osztály örökölje a viselkedését).
- 9. sor : Néhány példányváltozó deklarálása. A valódi értékeiket a 29 - 40. sorok metódusai (az eseménykezelők)fogják meghatározni.
- 11. sor : Az utasítás létrehoza az **IntVar()** osztály egy objektumát. Ez az osztály a *Tkinter* modul része ugyanúgy, mint a hasonló **DoubleVar()**, **StringVar()** és **BooleanVar()** osztályok. Ezek teszik lehetővé « *Tkinter változók* » definiálását, amik valójában objektumok, de változókként viselkednek a *Tkinter* widgetek belsejében.
  Így a **self.chk**-ban hivatkozott objektum egy egész típusú változó ekvivalensét tartalmazza a *Tkinter* által használható formában. Hogy a Pythonból hozzáférjünk az értékéhez, ennek az objekum osztálynak a specifikus metódusait kell használnunk : a **set()** metódussal rendelhetünk hozzá értéket, a **get()** metódussal kiolvashatjuk ki az értéket (amit a 47. sorban teszünk).
- 12. sor : A **checkbutton** objektum **variable** opciója az előző sorban definiált *Tkinter változóhoz* van asszociálva. (Egy *Tkinter* widget definíciójában nem tudunk közvetlenül hivatkozni egy közönséges változóra, mivel maga a *Tkinter* olyan nyelven van megírva, ami nem ugyanazokat a konvenciókat használja a változói formázására, mint a Python. A *Tkinter változóosztályokból* konstruált objektumok az interface biztosításához szükségesek.)
- 13. sor : A **command** opció azt a metódust adja meg, amit akkor kell a rendszernek hívni, amikor felhasználó az egérrel a checkboxba kattint.
- 14 - 24. sor : Ezek a sorok három hasonló utasításban definiálják a három cursor *widget*-et. Elegánsabb lenne ezeket egy programhurokban háromszor megismételt utasítással programozni. Ehhez azonban a *lambda* kifejezések fogalomára volna szükség, amit még nem magyaráztam el. A *widgetekkel* asszociált eseménykezelők definíciója is összetettebbé válna. Hagyjuk meg ez alkalommal a külön utasításokat, a későbbiekben törekedni fogok ennek a programrésznek a tökéletesítésére.

- 26. - 40. sorok : Az előző sorokban definiált 4 *widget* mindegyikének van egy **command** opciója. Mindegyik esetben eltérő a **command** opcióban hívott metódus : a checkbox a **setCurve()**, az első cursor a **setFrequency()**, a második cursor a **setPhase(),** a harmadik cursor a **setAmplitude()** metódust aktiválja. Jól jegyezzük meg, hogy a **Scale** *widgetek* **command** opciója az asszociált metódusnak átad egy argumentumot (az aktuális cursorpozíciót), míg ugyanez a **command** opció semmit sem ad át a **Checkbutton** *widget* esetében.

  Mind a négy metódus (amik a checkbox és a három cursor által előidézett események handlerei) az **event_generate()** metódus hívásával egy új esemény[53] megjelenését idézi elő.

  Ennek a metódusnak a hívásakor a Python az operációs rendszernek pontosan ugyanazt az üzenet-eseményt küldi, mint ami akkor keletkezne, ha a felhasználó egyszerre nyomná meg a <Ctrl>, <Shift> és <Z> billentyűket.
  Tehát egy speciális üzenet-eseményt hozunk létre, amit egy másik *widgethez* kapcsolt eseménykezelővel tudunk detektálni és kezelni (lásd a következő oldalt). Így ***valódi kommunikációs rendszert hozunk létre a widgetek között*** : minden alkalommal, amikor a felhasználó a vezérlő panelen végrehajt egy akciót, egy specifikus eseményt generál, ami figyelmeztet más *widget*-eket erre az akcióra.

  Megjegyzés *: más billentyűkombinációt is választhattunk volna (vagy éppen egy más eseménytípust). A választásunk azért esett erre, mert kevés az esélye annak, hogy a felhasználó ezt a billentyűkombinációt használja, mialatt a programunkat vizsgálja. Viszont mi magunk majd létre tudunk hozni ilyen teszteseményt a klaviatúrán, amikor eljön az ideje az eseménykezelő ellenőrzésének, amit egyéblént meg fogunk tenni.*

- 42 - 54 sorok : Mint az **oscillo.py** -t, ezt az új modult is kiegészítjük a főprogram szintjén néhány kódsorral. Ezek a sorok teszik lehetővé az osztály működésének tesztelését. Csak akkor hajtódnak végre, ha a modult mint önálló alkalmazást indítjuk el. Alkalmazza az olvasó is ezt a technikát a saját programjaiban, mert ez egy jó programozási gyakorlat. Az így megalkotott modulok alkalmazója (a modulok végrehajtása révén) nagyon könnyen (újra) felfedezheti azok funkcióit és (ennek a néhány kódsornak az elemezésével) a használatuk módját.

  Ezekben a tesztsorokban egy **root** főablakot hozunk létre, ami két widgetet tartalmaz : a **ChoiceVibra()** (rezgésválasztás) és a **Label()** osztály egy-egy widgetjét.

  Az 53. sorban a főablakhoz egy eseménykezelőt kapcsolunk : ettől kezdve minden specifikált típusú esemény az **showAll()** (mindent kiír) függvényt hívását idézi elő. Ez a függvény a speciális eseménykezelőnk, ami minden esetben, amikor az operációs rendszer egy <Shift-Ctrl-Z> típusú eseményt detektál végrahajtódik.

  Ahogyan föntebb már elmagyaráztam, úgy intéztük, hogy az ilyen eseményeket a **ChoiceVibra()** (rezgésválasztás) osztály objektumai hozzák létre, amikor a felhasználó módosítja a három cursor valamelyikének vagy a checkboxnak az állapotát.

53 Valójában ezt inkább egy üzenetnek kellene hívnunk (ami maga egy eseménynek a jelzése). Szíveskedjen újra olvasni e tárgykörben az ***Eseményvezérelt programok*** című fejezet magyarázatát a 86. oldalon.

Az **showAll()** (mindent kiír) függvény - mivel csak tesztelésre találtam ki – semmi mást nem csinál, csak a **Label()** osztály egy widgetjének **text** opcióját (újra)konfigurálva, a négy widgetünkkel összekapcsolt változók értékeit íratja ki.

- 47. sor : az **fra.chk.get()** kifejezés : föntebb láttuk, hogy a checkbox állapotát tároló változó egy *Tkinter* objektum-változó. *A* Python nem tudja közvetlenül olvasni az ilyen változó tartalmát, ami a valóságban egy interface objektum. Ahhoz, hogy az értéket megkapjuk, az osztály egy speciális metódusát kell használnunk : a **get()** metódust.

Az események terjedése

A fönt leírt kommunikációs mechanizmus tiszteletben tartja a *widget*-osztályok hierarchiáját. Jegyezzük meg, hogy az eseményt kiváltó metódus a **self**-fel ahhoz a *widget* -hez van kötve, aminek az osztályát éppen most definiáljuk. Általában egy üzenet-esemény egy bizonyos *widget*-hez van kötve (például egy gombra kattintás a gombhoz van kötve), ami azt jelenti, hogy az operációs rendszer először megvizsgálja, hogy van-e eseménykezelő erre az eseménytípusra, ami szintén ehhez a *widget*-hez van kapcsolva. Ha van ilyen, akkor ez az, ami aktiválva van és az üzenet terjedése leáll. Ha nincs, akkor az üzenet-esemény hierarchikus rendben egymás után jelenik meg a master *widget* -eknek mindaddig, míg egy eseménykezelőt nem talál, vagy el nem éri a főablakot.

A billentyűlenyomásoknak megfelelő események (amilyen a példában alkalmazott <Shift-Ctrl-Z>) viszont mindíg közvetlenül az alkalmazás főablakához vannak küldve. Példánkban ennek az eseménynek a handler-ét a **root** ablakhoz kell kapcsolnia.

Gyakorlatok :

13.12. Az ön widget-e a **Frame()** osztály tulajdonságait örökli. Tehát ennek az osztálynak az alapértelmezett opcióit a **configure()** metódussal módosítva megváltoztathatjuk a widget kinézetét. Próbáljon meg például úgy tenni, hogy a vezérlő panelt egy 4 pixel széles mélyedés vegye körül (*bd = 4, relief = GROOVE*). Ha nem érti, hogy mit kell tenni, merítsen ötletet az **oscillo.py** scriptből (10. sor).

13.13. Ha a **Scale()** widget-ek **showvalue** opciójának az 1 értéket adjuk, akkor állandóan ki lesz írva a cursor skálához viszonyított pontos pozíciója. Aktiváljuk ezt a funkciót a « fázis »-paramétert vezérlő cursor számára.

13.14. A **Scale()** widget-ek **troughcolor** opciója lehetővé teszi, hogy színt definiáljunk a csúszkáinknak. Használjuk ezt az opciót úgy, hogy a 3 cursor színe az új widget létrehozásakor paraméterként használt szín legyen.

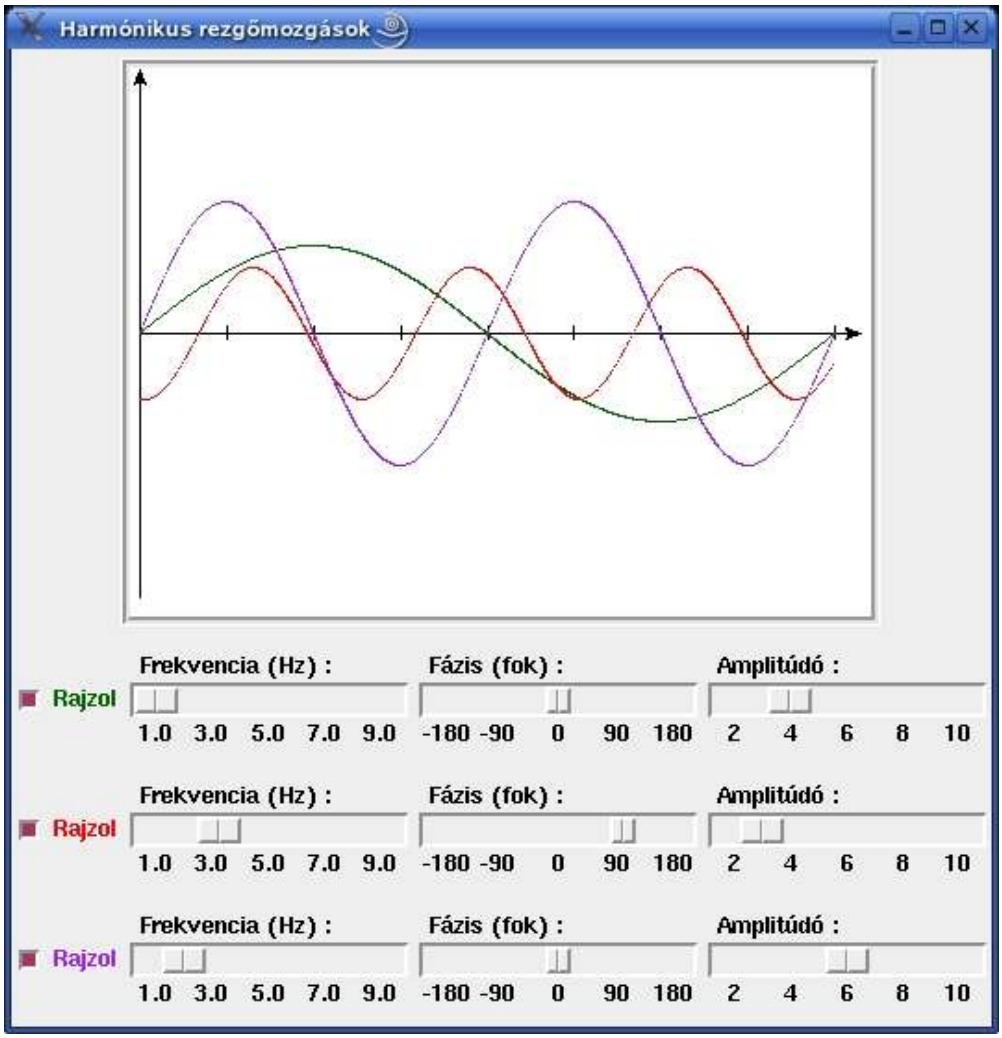
13.15. Módosítsa a scriptet úgy, hogy a cursor widget-ek távolabb legyenek egymástól (a **pack()** metódus **padx** és **pady** opciói).

13.5 A kompozit widget-ek beépítése egy összetett alkalmazásba

Az előző gyakorlatokban két új *widget*-osztályt hoztunk létre : a sinus-görbék kirajzolására specializált vásznat : az **OscilloGraphe()** *widget*-et; és a rezgés paramétereinek kiválasztását lehetővé tevő három cursoros vezérlőpanelt : a **ChoiceVibra()** (rezgésválasztás) *widget*-et.

A *widgetek* az **oscillo.py** és **curseurs.py**[54] modulokban rendelkezésünkre állnak.

Egy fizika órán bemutatható összetett alkalmazásban fogjuk őket felhasználni : egy **OscilloGraphe()** *widget* egy, két, vagy három különböző színű grafikont rajzol ki egymásra, melyek mindegyikét egy **ChoiceVibra()** (rezgésválasztás) *widget* vezérel :



A script a következő oldalon található.

Figyelje meg a rajzok frissítésére használt technikát ! Amikor a felhasználó a vezérlő panelek egyikén valamilyen akciót hajt végre egy « közvetítő » esemény keletkezik. Ez idézi elő a vásznon lévő rajz frissítését.

Emlékezzünk rá, hogy a grafikus interface-szel működő alkalmazásokat « eseményvezérelt programokként » kell fejleszteni (lásd 85. oldalt).

54 Magától értetődik, hogy az összes általunk konstruált osztályt egyetlen modulban egyesíthetnénk.

A példa előkészítésekor önkényesen döntöttem úgy, hogy a grafikon kiírását egy speciális esemény indítsa el. Ez tökéletesen hasonlít azokra az eseményekre, amiket az operációs rendszer akkor hoz létre, amikor a felhasználó valamilyen akciót hajt végre. A lehetséges események (igen széles) skálájáról egy olyat választottam, - a <Shift-Ctrl-Z> billentyű kombináció lenyomását - amit nem valószínű, hogy a felhasználó a program működése alatt más okból használni fog.

A **ChoiceVibra()** (rezgésválasztás) *widget*-osztályba beépítettem azokat az utasításokat, amik biztosítják, hogy - amikor a felhasználó megmozdítja az egyik cursort vagy megváltoztatja a checkbox állapotát - ilyen események jöjjenek létre. Definiálni fogom az eseményhez tartozó handlert és beépítem az új osztályunkba. **showCurves()** (megmutatja a görbét) lesz a neve és a grafikon frissítése lesz a feladata. Mivel az esemény típusa billentyűlenyomás, ezért azt az alkalmazás főablakának szintjén kell detektálni.

```
1.  from oscillo import *
2.  from curseurs import *
3.
4.  class ShowVibra(Frame):
5.      """Harmónikus rezgőmozgások bemutatása"""
6.      def __init__(self, boss =None):
7.          Frame.__init__(self)        # a szülő osztály constructora
8.          self.colour = ['dark green', 'red', 'purple']
9.          self.trace = [0]*3          # kirajzolandó görbék listája
10.         self.control = [0]*3       # kontrolpanelek listája
11.
12.         # vászonpéldány létrehozása az x és y koordináta-tengelyekkel :
13.         self.gra = OscilloGraphe(self, width_ =400, height_=200)
14.         self.gra.configure(bg ='white', bd=2, relief=SOLID)
15.         self.gra.pack(side =TOP, pady=5)
16.
17.         # 3 vezérlőpanel (cursorok) létrehozása :
18.         for i in range(3):
19.             self.control[i] = ChoiceVibra(self, self.colour[i])
20.             self.control[i].pack()
21.
22.         # Az ábrák kirajzolását indító események definiálása :
23.         self.master.bind('<Control-Z>', self.showCurves)
24.         self.master.title('Harmónikus rezgőmozgások
25.         self.pack()
26.
27.     def showCurves(self, event):
28.         """A három kitérés-idő grafikon (újra)kirajzolása """
29.         for i in range(3):
30.
31.             # Először töröljük az (esetleges) előző ábrát :
32.             self.gra.delete(self.trace[i])
33.
34.             # Aztán kirajzoljuk az új ábrát :
35.             if self.control[i].chk.get():
36.                 self.trace[i] = self.gra.drawCurve(
37.                                         colo = self.colour[i],
38.                                         freq = self.control[i].freq,
39.                                         phase = self.control[i].phase,
40.                                         ampl = self.control[i].ampl)
41.
42. #### Kód az osztály teszteléséhez : ###
43.
44. if __name__ == '__main__':
45.     ShowVibra().mainloop()
46.
```

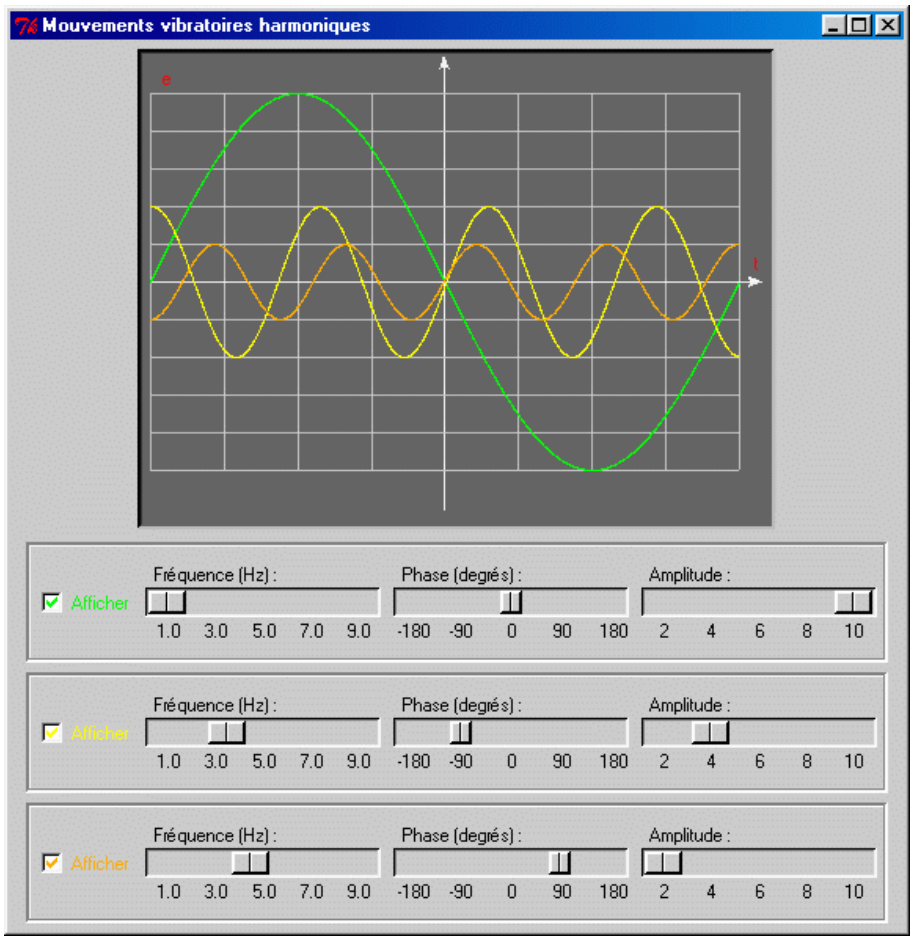
Magyarázatok :

- 1. - 2. sorok : Kihagyhatjuk a *Tkinter* modul importját, mert a két modul mindegyike gondoskodik róla.
- 4. sor : Már ismerjük a jó technikákat, ezért úgy döntöttem, hogy az alkalmazást egy **Frame()** osztályból leszármaztatott osztályként konstruálom meg. Így a későbbiekben teljes egészében beépíthetjük más projektekbe, ha úgy döntünk.
- 8. - 10. sorok : Néhány példányváltozó (3 lista) definíciója. A három kirajzolt görbe grafikus objektum. A színüket a **self.colour** listában definiálom. Készítenünk kell egy **self.trace** listát is a rajzok hivatkozásainak tárolására és egy **self.control** listát a három vezérlőpanel hivatkozásainak tárolására.
- 13. - 15. sorok : A rajzoló widget létrehozása. Mivel az **OscilloGraphe()** osztályt a **Canvas()** osztályból származtattam le, ezért a **Canvas()** osztály specifikus opcióinak újradefiniálásával mindíg lehetőség van a rajzoló widget konfigurálására (13. sor).
- 18. - 20. sorok : A három vezérlőpanelt egy programhurokban hozom létre. A hivatkozásaikat a 10. sorban a **self.control** listában tárolom. A vezérlőpaneleket az aktuális widget slave widgetjeiként hozom létre a **self** paraméter segítségével. A második paraméter a vezérlendő rajz színét adja meg.
- 23. - 24. sorok : Létrehozása pillanatában mindegyik *Tkinter* widget automatikusan kap egy master attribútumot, ami az alkalmazás főablakának hivatkozását tartalmazza. Ez a hivatkozás különösen hasznos, ha a főablakot a *Tkinter* implicit módon hozta létre, mint ebben az esetben.

  Emlékezzünk rá, amikor úgy indítunk el egy alkalmazást, hogy közvetlenül egy olyan widget-et hozunk létre, mint például a *Frame* (ezt tettük a 4. sorban), akkor a *Tkinter* automatikusan létrehoz ennek a widget- nek egy master ablakot (a **Tk()** osztály egy objektumát).
  Mivel ez az objektum automatikusan lett létrehozva, ezért a kódunkban a hozzáféréshez semmilyen hivatkozás sincs, erre csak a **master** attribútum révén van módunk, amit a *Tkinter* automatikusan társít minden widget-tel.
  Ezt a hivatkozást a főablak címsorának újradefiniálására használjuk (a 24. sorban) és arra, hogy egy eseménykezelőt kötünk hozzá (a 32. sorban).
- 27. - 40 . sorok : Ez a metódus a <Shift-Ctrl-Z> eseményeket kezeli, amiket mindíg akkor generálnak a **ChoiceVibra()** (rezgésválasztás) widget-jeink (vagy « vezérlő panel »-jeink), amikor a felhasználó a cursorral vagy a checkbox-szal csinál valamit. Először mindíg törli az esetleg jelen lévő rajzokat (32. sor) a **delete()** metódus segítségével. Az **OscilloGraphe()** widget szülőosztályától - a **Canvas()** osztálytól - örökölte ezt a metódust.
  Utána minden olyan vezérlőpanel esetén, melynek az « Rajzol » checkboxát bejelöltük, kirajzolja az új görbét. A vásznon így kirajzolt objektumok mindegyikének van egy hivatkozási száma, ami az **OscilloGraphe()** widgetünk **drawCurve()** (görbét rajzol) metódusának a visszatérési értéke.
  A rajzaink hivatkozási számait a **self.trace** listában tároljuk.
  Ezek teszik lehetővé a rajzok egyedi törölését (vesd össze a 32. sor utasításával).
- 36. - 40. sorok : A **drawCurve()** (görbét rajzol) metódusnak átadott frekvencia, fázis és amplitúdó értékek az egyes vezérlő panelek objektum-attribútumai, amiket a **self.control** listában tárolunk. Ezekhez az attribútumokhoz a pont operátorral történő minősített névmegadás segítségével férhetünk hozzá.

Gyakorlatok :

13.16. Módosítsa úgy a scriptet, hogy az alábbi kinézetet kapja (a képernyőn legyen referenciarács, a vezérlő panel mélyedéssel legyen körülvéve) :



13.17. Módosítsa úgy a scriptet, hogy az három helyett négy grafikus vezérlőt jelenítsen meg. A negyedik rajz színe legyen például : 'blue', 'navy', 'maroon', ..

13.18. A 38. - 40. sorokban a felhasználó által a kontrolpaneleken beállított frekvencia, fázis és amplitúdó értékeket közvetlenül a megfelelő példány-attribútumokból nyerjük ki. A Python megengedi ezt a praktikus megoldást, azonban ez egy veszélyes technika. Megsérti ugyanis az « objektum orientált programozás » általános elméletének egyik ajánlását : ***az objektumok tulajdonságait mindíg speciális metódusokkal kell kezelni***. Az ajánlás betartása érdekében adjunk a **ChoiceVibra()** (rezgésválasztás) osztályhoz egy **values()** -nek nevezett kiegészítő metódust, ami visszatérési értékként egy tuple-ban fogja megadni a választott frekvencia, fázis és amplitúdó értékeket. A scriptünk 38. - 40. sorait tehát valami olyasmivel kell helyettesíteni, mint a következő sor :

```
freq, phase, ampl = self.control[i].valeurs()
```

13.19. Írjon egy alkalmazást, ami megjelenítegy ablakot egy vászonnal és egy cursor widgettel (Scale). A vászonra rajzoljon egy kört, aminek a méretét a felhasználó a cursor segítségével tudja változtatni.

13.20. Írjon egy scriptet, ami két osztályt hoz létre : egy Frame()-ből leszármaztatott "Application" osztályt, aminek a constructora egy 400x400 pixeles vásznat fog létrehozni, valamint két

gombot. A vásznon a későbbiekben leírt "Visage" ("Arc") osztály egy objektumát fogjuk létrehozni
A "Visage" ("Arc") osztállyal egyszerűsített emberi arcot reprezentáló grafikai objektumokat definiálunk. Ezek egy nagy körből állnak, amiben három kisebb, a két szemet és a nyitott szájat reprezentáló ellipszis van. A "bezar" metódus a száj ellipszisét egy vízszintes szakasszal helyettesíti. A "kinyit" metódus állítja vissza a száj ellipszisét.
Az "Application" osztályban definiált két gomb fogja a vászonra tett Visage (Arc) objektum száját nyitni és zárnini..
(A 90. oldal példájából meríthet ötletet a kód egy részének megírásához).

13.21. Összegző gyakorlat : színszótár készítése.

Cél : egy olyan script készítése, amivel egyszerűen és gyorsan hozhatunk létre egy új színszótárat, mellyel bármelyik színhez a magyar nevével férhetünk hozzá.

Környezet : A különböző *Tkinter*-rel színezett objektumok kezelésekor megállapítottuk, hogy ez a grafikus könyvtár elfogadja, hogy a legalapvetőbb színeket az angol nevüket tartalmazó stringek formájában adjuk meg neki : 'red', 'blue', stb.

Azt is tudjuk, hogy a számítógép csak numerikus információkat tud kezelni. Ez azzal jár, hogy bármelyik színt előbb vagy utóbb szám formájában kell kódolni. Erre a kódolásra természetesen el kell fogadni egy konvenciót, ami rendszerről rendszerre változhat. Az egyik leggyakoribb konvenció szerint a színt három byte-tal reprezentáljuk, amik a vörös (**R**ed), a zöld (**G**reen) és a kék (**B**lue) komponensének intenzitását adják meg.

Ez a konvenció bármely színárnyalat előállítására alkalmazható a *Tkinter*-rel. Bármilyen grafikus elem színét megadhatjuk egy olyan 7 karakterből álló stringgel, mint a következő : "#00FA4E". A # karakter azt jelenti, hogy egy hexadecimális érték következik. A következő hat karakter az R, G, B színkomponensek 3 hexadecimális értékét reprezentálják. Egy szín és a kódja közötti megfeleltetés láthatóvá tételéhez ki lehet próbálni a **tkColorChooser.py** programot (ez általában a Python **/lib-tk** alkönyvtárában található meg).

Mivel nehéz megjegyezni a hexadecimális kódokat, ezért a *Tkinter*-nek van egy konverziós szótára, ami lehetővé teszi, hogy a leggyakoribb színek közül néhánynak az angol nevét használjuk.

A cél egy olyan program készítése, ami meg fogja könnyíteni egy ekvivalens magyar szótár kivitelezését, amit majd be építhet a programjaiba. Amikor elkészült, a szótár a következő formájú lesz : {'zöld':'#00FF00', 'kék':'#0000FF', ... stb ...}.

Feladat meghatározás :

Egy osztályra épülő grafikus alkalmazást kell megvalósítani.

Ez egy ablakból, adatbeviteli mezőkből és gombokból fog állni, hogy a felhasználó könnyen kódolhassa az új színeket : az egyik mezőben a szín magyar nevét, a másik mezőben a hexadecimális kódját kell megadni.

Amikor a szótárban már vannak adatok, akkor tesztelhetőnek kell lennie, vagyis be kell

tudni írni a szín magyar nevét és egy gomb segítségével meg kell tudni találni a megfelelő hexadecimális kódot (esetleg egy színes csíkot írathat ki).
Egy gombbal lehessen szövegfile-ba menteni a szótárat. Egy másik gombbal lehessen rekonstruálni a szótárat egy szövegfile-ból.

13.22. Az alábbi script a képernyőre különböző módon elhelyezett játékkocka csoportokat rajzoló projekt vázlata (ez a vázlat az első fázisaa lehet egy játékprogram megvalósításának).
A gyakorlat a script elemzéséből és kiegészítéséből áll. Bele kell képzelnie magát annak a programozónak a helyébe, akinek folytatnia kell egy másik programozó munkáját, vagy egy olyan informatikus helyzetébe, akit felkértek, hogy vegyen részt egy csapat munkájában.

A) Kezdje a script elemzésével és írjon hozzá kommenteket, főleg a #*** -gal jelölt sorokhoz, hogy megmutassa, megértette mit kell a programnak tenni a jelzett helyeken :

```
from Tkinter import *

class FaceDom:
    def __init__(self, can, val, pos, size =70):
        self.can =can
        # ***
        x, y, c = pos[0], pos[1], size/2
        can.create_rectangle(x -c, y-c, x+c, y+c, fill ='ivory', width =2)
        d = size/3
        # ***
        self.pList =[]
        # ***
        pDispo = [((0,0),), ((-d,d),(d,-d)), ((-d,-d), (0,0), (d,d))]
        disp = pDispo[val -1]
        # ***
        for p in disp:
            self.circle(x +p[0], y +p[1], 5, 'red')

    def circle(self, x, y, r, colo):
        # ***
        self.pList.append(self.can.create_oval(x-r,y-r,x+r,y+r, fill=colo))

    def erase(self):
        # ***
        for p in self.pList:
            self.can.delete(p)

class Project(Frame):
    def __init__(self, width_, height_):
        Frame.__init__(self)
        self.width_, self.height_ = width_, height_
        self.can = Canvas(self, bg='dark green',width=width_,height=height_)
        self.can.pack(padx =5, pady =5)
        # ***
        bList = [("A", self.buttonA), ("B", self.buttonB),
                 ("C", self.buttonC), ("D", self.buttonD),
                 ("Quit", self.buttonQuit)]
        for b in bList:
            Button(self, text =b[0], command =b[1]).pack(side =LEFT)
        self.pack()

    def buttonA(self):
        self.d3 = FaceDom(self.can, 3, (100,100), 50)
```

```
    def buttonB(self):
        self.d2 = FaceDom(self.can, 2, (200,100), 80)

    def buttonC(self):
        self.d1 = FaceDom(self.can, 1, (350,100), 110)

    def buttonD(self):
        # ***
        self.d3.erase()

    def buttonQuit(self):
        self.master.destroy()

Project(500, 300).mainloop()
```

B) Módosítsa a scriptet úgy, hogy megfeleljen a következő feladat meghatározásnak :

- Legyen a vászon nagyobb : 600 x 600 pixel méretű.
- Az utasításgombokat jobboldalra kell áthelyezni és nagyobb távolságot kell hagyni.
- A dobókocka felszínén lévő pontok méretének a felszínnel arányosan kell változni.

1. változat : Csak 2 gombot hagyjon meg : az A-t és a B-t. Az A gombra történő minden egyes kattintásra három új (azonos méretű, inkább kicsi), egy oszlopban elhelyezkedő kocka jelenjen meg. A kockákon a pontok száma véletlenszerűen essen 1 és 6 közé. Minden új oszlop az előzőtől jobbra legyen elhelyezve. Ha a 3 kockán a pontok (bármilyen sorrendben) 4, 2. 1 -nek felelnek meg, akkor a « nyert » szöveget kell kiírni az ablakba (vagy a vászonra). A B gomb az összes kirajzolt dobókockát (nemcsak a pontokat !) törli.

2. változat : Csak 2 gombot hagyjon meg : az A-t és a B-t. Az A gomb 5 dobókockát rajzoltasson ki. A kockák úgy legyenek elrendezve, mint az 5-ös pontérték pontjai. A véletlenszerűen sorsolt értékek 1 és 6 közé essenek és minden érték csak egyszer forduljon elő. A B gomb az összes kirajzolt dobókockát (nemcsak a pontokat !) törölje.

3. változat : Csak 3 gombot hagyjon meg : az A, B és C-t. Az A gomb kör alakban elrendezve 13 azonos méretű dobókockát rajzoltasson ki. Minden kattintás a B gombon változtassa először meg az első, majd a második, harmadik, stb. kocka pontértékét. A új pontértékek eggyel nagyobbak legyenek, mint az előző érték, kivéve ha az előző érték 6 volt : ez esetben az új érték 1 legyen. A C gomb az összes kirajzolt dobókockát (nemcsak a pontokat !) törölje.

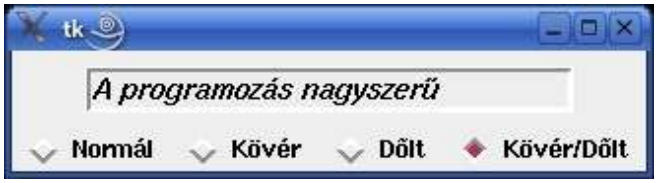
4. változat : Csak 3 gombot hagyjon meg : az A, B és C-t. Az A gomb 12 azonos méretű kockát rajzoltasson ki két sorba, soronként 6-ot. Az első sorban a pontértékek sorrendje 1, 2, 3, 4, 5, 6 legyen. Minden kattintás a B gombon a második sor első kockájának pontértékét mindaddig véletlenszerűen változtassa meg, amíg az új érték az első sorbeli megfelelő kocka pontértékétől eltér. Amikor a 2. sor 1. kockájának pontértéke az 1. sorbeli megfelelőjének a pontértéke lesz, akkor a 2. sor 2. kockájának pontértéke változzon véletlenszerűen, és így tovább, egészen addig, amíg az alsó 6 kocka pontértékei meg nem egyeznek a felső kockák pontértékeivel. A C gomb az összes kirajzolt dobókockát (nemcsak a pontokat !) törölje.

14. Fejezet : És még néhány widget ...

A következő oldalakon található kiegészítő utalások és példák hasznosak lehetnek az olvasó saját projekteinek fejlesztésekor. Nyilván nem egy teljes *Tkinter* referencia dokumentációról van szó. Ha az olvasó többet akar tudni, akkor előbb vagy utóbb speciális könyveket kell megnéznie, mint amilyen például John E. Grayson kitűnő könyve a : *Python and Tkinter programming. A könyv* teljes hivatkozása megtalálható a 8. oldalon az irodalomjegyzékben.

14.1 A rádiógombok

A « rádiógomb » widgetek egymást kölcsönösen kizáró lehetőségek felkínálását teszik lehetővé a felhasználónak. A régi rádiókészülékek sávváltó gombjaival való analógia miatt kapták nevüket. Ezeket a gombokat úgy szerkesztették meg, hogy egyszerre csak egyet lehetett benyomni, az összes többi automatikusan kiugrott.



Ezeknek a widgeteknek lényeges jellemzője, hogy mindíg csoportokban használjuk őket. Az azonos csoporthoz tartozó gombok ugyanahhoz a *Tkinter változóhoz* vannak kötve, de mindegyikükhöz külön érték van rendelve.
Amikor a felhasználó kiválasztja az egyik gombot, akkor a gombhoz rendelt érték lesz hozzárendelve a közös *Tkinter változóhoz.*

```
1.   from Tkinter import *
2.
3.   class RadioDemo(Frame):
4.       """Demó : a rádiógomb widgetek használata"""
5.       def __init__(self, boss =None):
6.           """Egy adatbeviteli mező és 4 rádiógomb létrehozása"""
7.           Frame.__init__(self)
8.           self.pack()
9.           # Szöveget tartalmazó adatbeviteli mező :
10.          self.texte = Entry(self, width =30, font ="Arial 14")
11.          self.texte.insert(END, "A programozás nagyszerű")
12.          self.texte.pack(padx =8, pady =8)
13.          # A 4 betűstílus magyar és technikai neve :
14.          styleFontHu =["Normál", "Kövér", "Dőlt", "Kövér/Dőlt"]
15.          styleFontTk =["normal", "bold", "italic"  , "bold italic"]
16.          # Az aktuális stílust egy Tkinter 'változó-objektumba mentjük' ;
17.          self.choiceFont = StringVar()
18.          self.choiceFont.set(styleFontTk[0])
19.          # A 4 rádiógomb létrehozása' :
20.          for n in range(4):
21.              bout = Radiobutton(self,
22.                                 text = styleFontHu[n],
23.                                 variable = self.choiceFont,
24.                                 value = styleFontTk[n],
25.                                 command = self.changeFont)
26.              bout.pack(side =LEFT, padx =5)
27.
28.      def changeFont(self):
29.          """Az aktuális betűstílus helyettesítése"""
30.          font_ = "Arial 15 " + self.choiceFont.get()
31.          self.texte.configure(font =font_)
32.
33.  if __name__ == '__main__':
34.      RadioDemo().mainloop()
```

Magyarázatok :

- 3. sor : Alkalmazásunkat most is inkább a **Frame()** osztályból leszármaztatott osztályként konstruáljuk meg. Ez lehetővé teszi, hogy esetleg egy nagyobb alkalmazásba nehézségek nélkül beépíthessük.
- 8. sor : Általában a widgetek létrehozása után alkalmazzuk a **pack()**, **grid()**, vagy **place()** pozícionáló metódusokat, amik a widgetek master-ablakbeli pozíciójának szabad megválasztását teszik lehetővé. Azonban lehetőség van arra is, hogy a widget constructorában előre gondoskodjunk a pozícinálásról. Ezt is bemutatom.
- 11. sor : Az **Entry** osztály widgetjeinek több metódusa van, amikkel hozzá lehet férni a beírt stringhez. A **get()** metódussal kapjuk vissza az egész stringet. A **delete()** -tel törölhetjük az egész stringet, illetve egy részét (vesd össze a « Színkódok » projekttel : 200. oldal). Az **insert()** -tel bármelyik pozícióba új karaktereket szúrhatunk be (vagyis egy esetleg már létező string elejére, végére, vagy a belsejébe). Ezt a metódust két argumentummal használjuk, az első megadja a beszúrás helyét (0-t kell megadni, ha a string elejére, END-et, ha a végére, vagy valamilyen numerikus indexet, ha egy közbenső pozícióba akarunk beszúrni).
- 14 .-15. sorok : A négy gombot nem külön utasításokkal, hanem inkább egy ciklussal hozzuk létre. Ezt megelőzően a gombok speciális opcióit a **styleFontHu** (fontstílus) és a **styleFontTk** listákban tároljuk : az előbbiben a gombok mellé kiírandó szövegeket, az utóbbiban a gombokhoz rendelendő értékeket tároljuk.
- 17 .-18. sorok : Ahogyan az előző oldalon magyaráztam, a négy gomb egy közös változó körül alkot csoportot. Ez a változó a felhasználó által választott rádiógombhoz rendelt értéket fogja felvenni. Erre a szerepre azonban nem használhatunk közönséges változót, mert a *Tkinter* objektumok belső attribútumai csak speciális metódusokkal férhetők hozzá. Egy string típusú *Tkinter* objektum-változót használunk itt, amit a **StringVar()** osztályból hozunk létre, és aminek a 18. sorban egy alapértelmezett értéket adunk.
- 20 .-26. sorok : Létrehozzuk a négy rádiógombot. Mindegyikhez különböző címkét és értéket rendelünk, de mindegyik gomb ugyanahhoz a közös *Tkinter változóhoz* (**self.choiceFont**) (választott font) van kapcsolva. Amikor a felhasználó az egérrel rákattint valamelyikre, akkor ugyanannak a **self.changeFont()** (font megváltoztatása) metódusnak a hívására kerül sor.
- 28. -31. sor : A betűtípus megváltoztatása az **Entry** widget **font** opciójának az újrakonfigurálásával történik. Ez az opció a betűtípus nevét, méretét és esetleg stílusát tartalmazó tuple-t vár. Ha a betűtípus neve nem tartalmaz betűközt, a tuple-t egy stringgel lehet helyettesíteni.

Példák :

```
('Arial', 12, 'italic')
('Helvetica', 10)
('Times New Roman', 12, 'bold italic')
"Verdana 14 bold"
"President 18 italic"
```

Lásd a . oldal példáit is.

14.2 Ablak összeállítása keretekből (frame-ekből)

A **Frame()** widget-osztályt már sokszor használtuk új, összetett widgetek létrehozására leszármaztatással.

A következő script bemutatja, hogy hogyan használható ez az osztály widget-csoportok létrehozására és az utóbbiaknak egy ablakban meghatározott módon való elrendezésére. Bizonyos dekoratív opciók (szegélyek, kiemelkedés, stb.) használatát is bemutatja.

A szemközti ablak elkészítéséhez két keretet - **f1** és **f2** – használtam úgy, hogy két külön widget-csoportot képezzenek : az egyiket a bal-, a másikat a jobboldalon. Úgy színeztem a kereteket, hogy nyilvánvalóvá tegyem ezt, de a színezés nyilván nem feltétlenül szükséges.

Az **f1** keret 6 másik keretet tartalmaz, melyek mindegyike a **Label()** osztály egy widgetjét tartalmazza. Az **f2** keret egy **Canvas()** és egy **Button()** widgetet tartalmaz. A színek és a dekorációk egyszerű opciók.

```
1.   from Tkinter import *
2.
3.   ablak = Tk()
4.   ablak.title("Keretekkel létrehozott ablak")
5.   ablak.geometry("300x300")
6.
7.   f1 = Frame(ablak, bg = '#80c0c0')
8.   f1.pack(side =LEFT, padx =5)
9.
10.  fint = [0]*6
15.  for (n, col, rel, txt) in [(0, 'grey50', RAISED, 'Kiemelkedő felület'),
16.                             (1, 'grey60', SUNKEN, 'Bemélyedő felület'),
11.                             (2, 'grey70', FLAT, 'Sík felület'),
12.                             (3, 'grey80', RIDGE, 'Gerinc'),
13.                             (4, 'grey90', GROOVE, 'Árok'),
14.                             (5, 'grey100', SOLID, 'Szegély')]:
15.      fint[n] = Frame(f1, bd =2, relief =rel)
16.      e = Label(fint[n], text =txt, width =15, bg =col)
17.      e.pack(side =LEFT, padx =5, pady =5)
18.      fint[n].pack(side =TOP, padx =10, pady =5)
19.
20.  f2 = Frame(ablak, bg ='#d0d0b0', bd =2, relief =GROOVE)
21.  f2.pack(side =RIGHT, padx =5)
22.
23.  can = Canvas(f2, width =80, height =80, bg ='white', bd =2, relief =SOLID)
24.  can.pack(padx =15, pady =15)
25.  gomb =Button(f2, text='Gomb')
26.  gomb.pack()
27.
28.  ablak.mainloop()
```

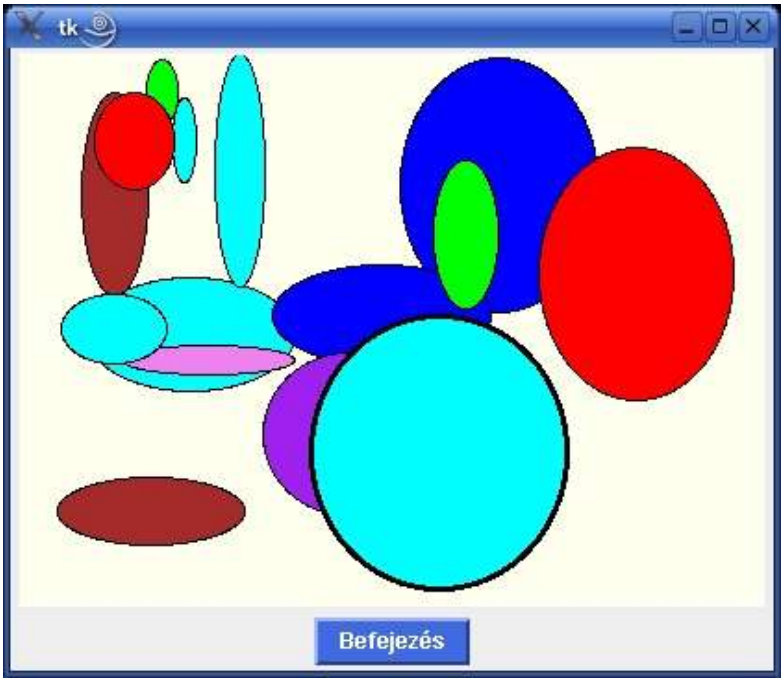
- 3.-5. sorok : A bemutatót a maximális egyszerűség érdekében nem fogom új osztályként programozni. Vegyük észre az 5. sorban a főablak méreteinek rögzítésére szolgáló **geometry()** metódus alkalmazását.
- 7. sor : A baloldali keretet hozzuk létre. A háttérszínt (a ciánkék egy változatát) a **bg** (*background*) argumentum határozza meg. Ez a string hexadecimális írásmódban tartalmazza annak a színnek a piros, zöld és kék komponensét, amit szeretnénk beállítani. A **#** karakter után, ami azt jelenti, hogy egy hexadecimális érték következik, hat alfanumerikus szimbólumot találunk. Ezek három darab kétjegyű hexadecimális számnak felelnek meg, melyek mindegyike egy 1 és 255 közé eső decimális értéket reprezentál. Így a **80**-nak a decimális128 és a **c0**-nak a decimális192 érték felel meg. Példánkban a piros, zöld és kék színkomponensek decimális értékei rendre : 128, 192 és 192. .
  Ezt a leíró technikát alkalmazva a fekete : **#000000**, a fehér : **#ffffff**, a piros : **#ff0000**, a sötétkék : **#000050**, stb.
- 8. sor : Mivel a **pack()** metódust alkalmazzuk, ezért a keret méreteit automatikusan határozza meg annak tartalma. A **side =LEFT** opció a keretet balra fogja pozícionálni a master ablakban. A **padx =5** opció jobb- és baloldalon egy 5 pixel széles sávot fog szabaddá tenni (« padx » -et fordíthatjuk « vízszintes távolságnak » is).
- 10. sor : Az **f1** keretben hat másik keretet, - melyek mindegyike egy címkét tartalmaz - akarunk csoportba foglalni. A kód egyszerűbb és hatékonyabb lesz, ha ezeket a widgeteket egy listában hozzuk létre, nem pedig független változókban. Készítünk egy 6 elemű listát, amit majd később töltünk fel.
- 11. - 16. sor : A 6 hasonló keret elkészítéséhez egy 6 tuple-ből álló listát kell bejárnunk. A tuple-k az egyes keretek jellemzőit tárolják. Minden tuple 4 elemből áll: egy indexből, egy a keret kiemelkedését definiáló *Tkinter* konstansból és két stringből, amik a címke színét és szövegét írják le.
  A **for** ciklus 6 iterrációt hajt végre a 6 elemű lista bejárásához. Mindegyik iterrációban az **n**, **col, rel** és **txt** változókhoz hozzárendeli az egyik tuple tartalmát (és utána végrehajtja a 17.-20. sorokat). Egy tuple-kből álló lista **for** ciklussal való bejárása rendkívül tömör konstrukciót alkot, ami nagyon kisszámú utasítással sok értékadást tesz lehetővé.
- 17. sor : A 6 keretet a **fint** lista elemeiként hozza létre. Mindegyiket egy két pixel széles, kiemelkedés hatását keltő dekoratív szegély díszíti.
- 18. - 20. sorok : A címkék mind azonos méretűek, de a szövegük és a háttérszínük különböző. A **pack()** metódus alkalmazása miatt a címkék mérete az, ami meghatározza a kis keretek méretét. Ez utóbbiak pedig az őket csoportba foglaló keret (**f1**) méretét határozzák meg. A **padx** és **pady** opciók teszik lehetővé, hogy egy kis távolságot tartsunk minden címke és minden kis keret körül. A **side =TOP** opció a 6 kis keretet egymás fölött helyezi el az **f1** konténerkeretben.
- 22. - 23. sorok : Az **f2** keretet (jobboldali keret) készítjük el. A színe egy sárga színárnyalat lesz és egy bemélyedés benyomását keltő szegéllyel vesszük körül.
- 25. - 28. sorok : Az **f2** keret egy vásznat és egy gombot fog tartalmazni. Megint figyeljük meg a widgetek körüli üres sáv biztosítására szolgáló **padx** és **pady** opciók használatát. (Nézzük meg például azt a gombot, amire nem alkalmaztuk ezt az opciót : emiatt a gomb hozzáér a keret széléhez.) Ahogyan a keretekkel tettem, ugyanúgy a vászon körül is elhelyeztem egy szegélyt. Tudjunk róla, hogy más widgetek – gombok, adatbeviteli mezők, stb. - is hasonlóan díszíthetők.

14.3 Hogyan mozgassunk az egérrel rajzokat

A *Tkinter* grafikus könyvtár egyik erőssége a vászon widget. Számos, igen hatékony eszköz van beleépítve a rajzok kezelésére. A következő scriptben, néhány alaptechnikát akarok bemutatni. Ha az olvasó többet akar ezekről megtudni, nevezetesen a több részből álló rajzok manipuálása vonatkozásában, akkor nézze meg valamelyik *Tkinter*-rel foglalkozó referenciaművet.

Alkalmazásunk az indításakor véletlenszerűen készít néhány rajzot a vásznon (néhány színes ellipszist). Az egérrel megragadva bármelyiküket el tudjuk mozgatni.

Amikor egy rajzot mozgatunk, akkor az a többi rajz síkja elé jön és a széle vastagabb lesz a mozgatás alatt.



Az alkalmazott technika megértéséhez emlékeznünk kell arra, hogy egy grafikus interface-t használó program « esemény vezérelt program » (ha magyarázatra van szüksége nézze át a 85. oldalon leírtakat) Az alkalmazásba egy olyan mechanizmust fogok beépíteni, ami reagál a : « bal egérgomb lenyomása », « egér mozgatása lenyomott bal egérgombbal », « bal egérgomb elengedése » eseményekre.

Ezeket az eseményeket az operációs rendszer hozza létre és a *Tkinter* interface kezeli. A programozás abból fog állni, hogy ezeket az eseményeket különböző eseménykezelőkhöz (függvényekhez vagy metódusokhoz) kapcsoljuk.

```
# A példa bemutatja, hogyan kell ahhoz eljárnunk, hogy az egérrel a vászonra
# rajzolt objektumokat manipulálni tudjunk

from Tkinter import *
from random import randrange

class Draw(Frame):
    "A program főablakát definiáló osztály"
    def __init__(self):
        Frame.__init__(self)
        # A vászon létrehozása – 15 színes ellipszis rajzolása :
        self.c = Canvas(self, width =400, height =300, bg ='ivory')
        self.c.pack(padx =5, pady =3)
        for i in range(15):
            # Véletlenszerűen kisorsolunk egy színt :
            coul =['brown','red','orange','yellow','green','cyan','blue',
                   'violet', 'purple'][randrange(9)]
            # Véletlen koordinátájú ellipszis rajzolása :
            x1, y1 = randrange(300), randrange(200)
            x2, y2 = x1 + randrange(10, 150), y1 + randrange(10, 150)
            self.c.create_oval(x1, y1, x2, y2, fill =coul)
        # Az <egéresemények> hozzákapcsolása a <canevas> (vászon) widget-hez :
        self.c.bind("<Button-1>", self.mouseDown)
        self.c.bind("<Button1-Motion>", self.mouseMove)
        self.c.bind("<Button1-ButtonRelease>", self.mouseUp)
        # A kilépésgomb létrehozása :
        b_fin = Button(self, text ='Befejezés', bg ='royal blue', fg ='white',
                       font =('Helvetica', 10, 'bold'), command =self.quit)
        b_fin.pack(pady =2)
        self.pack()

    def mouseDown(self, event):
        "Balegérgomb lenyomására végrehajtandó művelet"
        self.currObject =None
        # event.x és event.y tartalmazzák a kattintás koordinátáit :
        self.x1, self.y1 = event.x, event.y
        # <find_closest> a legközelebbi rajz referenciáját adja meg :
        self.selObject = self.c.find_closest(self.x1, self.y1)
        # Módosítjuk a rajz körvonalának a vastagságát :
        self.c.itemconfig(self.selObject, width =3)
        # <lift> átviszi a rajzot az előtérbe :
        self.c.lift(self.selObject)

    def mouseMove(self, event):
        "Lenyomott balgombbal mozgó egérrel végrehajtandó művelet"
        x2, y2 = event.x, event.y
        dx, dy = x2 -self.x1, y2 -self.y1
        if self.selObject:
            self.c.move(self.selObject, dx, dy)
            self.x1, self.y1 = x2, y2

    def mouseUp(self, event):
        "A balegérgomb fölengedésekor végrehajtandó művelet"
        if self.selObject:
            self.c.itemconfig(self.selObject, width =1)
            self.selObject =None

if __name__ == '__main__':
    Draw().mainloop()
```

Magyarázatok :

A script lényegében egy **Frame()**-ből származtatott grafikus osztály definícióját tartalmazza.

A script fő része, amint az az objektumokat használó programok esetében gyakran előfordul, egyetlen összetett utasítás. Ebben két egymást követő műveletet végzünk : létre hozzuk az előzőleg definiált osztály egy objektumát és aktiváljuk a **mainloop()** metódusát (ami elindítja az eseményfigyelőt).

A **Draw()** osztály constructorának struktúrája már ismerős kell, hogy legyen : a szülő osztály constructorát hívja, majd különböző widgeteket hoz létre.

A vászon-widgeten 15 rajzot hozunk létre, de nem foglalkozunk azzal, hogy a hivatkozásaikat változókban tároljuk. Azért járhatunk így el, mert a *Tkinter* egy belső hivatkozást őriz meg minden egyes objektum számára. (Ha más grafikus könyvtárakkal dolgozunk, valószínűleg gondoskodnunk kell ezeknek a hivatkozásoknak a tárolásáról.)

A rajzok színes ellipszisek. A színüket egy 9 színt tartalmazó listából véletlenszerűen választjuk ki, ugyanis a kiválasztott szín indexe a **random** modulból importált **randrange()** függvénnyel van meghatározva.

Ezután az interakciós mechanizmus beállítása következik : a vászon-widgethez tartozó **<Button-1>**, **<Button1-Motion>** és **<Button1-ButtonRelease>** esemény azonosítókat összekapcsoljuk a három eseménykezelő metódus nevével. (Ezeket a neveket mi választjuk az eseménykezelő metódusoknak.)

Amikor a felhasználó lenyomja a bal egérgombot, a **mouseDown()** metódus aktiválódik és az operációs rendszer egy eseményobjektumot ad neki át argumentumként, aminek **x** és **y** attribútumai tartalmazzák az egér koordinátáit a vászonra kattintás pillanatában.

Ezeket a koordinátákat közvetlenül a **self.x1** és **self.x2** objektumváltozókba tároljuk, mert máshol szükségünk lesz rájuk. Ezt követően a vászon-widget **find_closest()** metódusát alkalmazzuk, ami a legközelebbi rajz hivatkozását adja meg visszatérési értékként. (Megjegyzés : ez a praktikus metódus mindig visszaad egy hivatkozást, még akkor is, ha nem kattintottunk a rajz belsejébe.)

A többi már egyszerű : a kiválasztott rajz hivatkozását eltároljuk egy példányváltozóba és hívhatjuk a vászon-widget más metódusait a rajz jellemzőinek módosítására. Az **itemconfig()** és **lift ()** metódusokat használjuk a rajz körvonalának véknyítására és a rajz előtérbe hozására.

A rajz « szállítását » a **mouseMove()** metódus végzi, ami minden alkalommal végrehajtódik, amikor az egér mozog és a bal egérgomb le van nyomva. Az **event** objektum ez alkalommal is az egércursor koordinátáit tartalmazza a mozgás végén. Ezt használjuk fel az új és az előző koordináták közötti különbség kiszámítására, amit a vászon-widget mozgatást végző **move()** metódusának paraméterként adunk át.

Ezt a metódust azonban csak akkor tudjuk hívni, ha valóban ki van választva egy objektum és arra is ügyelnünk kell, hogy mentsük az új koordinátákat.

A **mouseUp()** metódus fejezi be a műveletet. Amikor a mozgatott rajz megérkezik a rendeltetési helyére, törölni kell a kiválasztást és a rajz körvonalának vastagságát az eredetire kell visszaállítani. Természetesen ez csak akkor képzelhető el, ha valóban van egy kiválasztás.

14.4 Python Mega Widgetek

A ***Pmw*** modulok a *Tkinter* érdekes kiterjesztései. Teljes egészükben Pythonban vannak megírva, valamennyien tartalmazzák a *Tkinter* alaposztályokból konstruált összetett widgeteknek egy könyvtárát. Ezek a rendkívül sokrétű funkcionalitással ellátott widgetek összetett alkalmazások gyors fejlesztésekor nagyon értékesek lehetnek. Ha használni akarjuk őket, tudjunk róla, hogy a *Pmw* modulok nem képezik részét a standard Python telepítésnek. Ezért mindíg ellenőriznünk kell, hogy rajta vannak-e azon a gépen, amin a programjaink futni fognak.

Nagyszámú mega-widget létezik. Csak néhányat fogok bemutatni a leghasznosabbak közül. A mega-widgeteket kísérő demo-scriptek kipróbálásával gyorsan fogalmat alkothatunk sokféle alkalmazási lehetőségükről (indítsuk például el a .../Pmw/demos könyvtárban lévő **all.py** scriptet).

14.4.1 « Combo Box »

A mega-widgeteket könnyű használni. A következő kis alkalmazás bemutatja, hogyan lehet egy **ComboBox** típusú widgetet (egy adatbeviteli mezővel kombinált adatlistát) használni. A legmegszokottabb módon konfiguráltam (egy legördülő listával).

Amikor a felhasználó a legördülő listából egy színt választ (illetve közvetlenül be is írhatja egy szín nevét az adatbeviteli mezőbe), ez a szín automatikusan a master ablak háttérszínévé válik.

Ebben a master ablakban elhelyeztem egy címkét és egy gombot, hogy megmutassam, hogyan lehet hozzáférni a C*omboBox* -ban előzőleg eszközölt kiválasztáshoz (a gombra kattintás az utoljára választott szín kiíratását fogja előidézni).

```
1.   from Tkinter import *
2.   import Pmw
3.
4.   def changeColo(col):
5.       fen.configure(background = col)
6.
7.   def changeLabel():
8.       lab.configure(text = combo.get())
9.
10.  colours = ('navy', 'royal blue', 'steelblue1', 'cadet blue',
11.              'lawn green', 'forest green', 'dark red',
12.              'grey80','grey60', 'grey40', 'grey20')
13.
14.  ablak = Pmw.initialise()
15.  button_ = Button(ablak, text ="Test", command =changeLabel)
15.  button_.grid(row =1, column =0, padx =8, pady =6)
15.  lab = Label(ablak, text ='néant', bg ='ivory')
15.  lab.grid(row =1, column =1, padx =8)
16.
15.  combo = Pmw.ComboBox(ablak, labelpos = NW,
17.                       label_text = 'Válasszon színt :',
18.                       scrolledlist_items = colours,
19.                       listheight = 150,
20.                       selectioncommand = changeColo)
21.  combo.grid(row =2, columnspan =2, padx =10, pady =10)
22.
15.  ablak.mainloop()
```

Magyarázatok:

- 1. és 2. sor : A szokásos *Tkinter* komponensek és a **Pmw** modul importjával kezdünk.
- 14. sor : A master ablak létrehozásához inkább a **Pmw.initialise()** metódust kell használni, nem pedig közvetlenül a **Tk()** osztályból létrehozni egy objektumot. Ez a metódus gondoskodik arról, hogy mindaz installálva legyen, ami ahhoz szükséges, hogy amikor ezt az ablakot megszüntetjük, akkor a slave widgetjeinek a megszüntetése korrekten történhessen meg. Ez a metódus egy jobb hibaüzenet kezelőt is installál.
- 20. sor : A **labelpos** opció az adatbeviteli mezőt kísérő címke elhelyezkedését határozza meg. Példánkban a mező fölött helyeztük el. Máshol is elhelyezhetnénk, például a mezőtől balra (**labelpos = W**). Jegyezzük meg, hogy ez az opció nélkülözhetetlen, ha címkére van szükségünk (alapértelmezett értéke nincs).
- 24. sor : A **selectioncommand** opció egy argumentumot ad át a hívott függvénynek : a listbox-ban kiválasztott elemet. Ezt a választást a **get()** metódussal is meg lehet találni, ahogyan a 8. sorban tesszük a címke aktualizálása érdekében.

14.4.2 Ékezetes karakterek beírására vonatkozó megjegyzés

Az előzőekben már említettem, hogy a Python képes a világ valamennyi abc-jét (görög, ciril, arab, japán, stb. lásd a 41. oldalt) kezelni. Ugyanez áll fönn a *Tkinter* -re. Magyar anyanyelvűként biztos, hogy szeretnénk, ha a scriptjeink felhasználói tudnának ékezetes karaktereket beírni az *Entry*, *Text widgetekbe* és leszármazottaikba (*ComboBox*, *ScrolledText*).

Jegyezzük meg, hogy amikor ezeknek a *widgeteknek* az egyikébe beírunk egy vagy több nem-ASCII karaktert tartalmazó stringet (például egy ékezetes karaktert), a Tkinter ezt a stringet az UTF-8 norma szerint kódolja. Ha a számítógépünk alapértelmezetten inkább a Latin-1 kódolást használja (leggyakrabban ez a helyzet), akkor a kiíratás előtt át kell konvertálni a stringet.

Ez a beépített **encode()** függvénnyel nagyon könnyen megtehető. Példa :

```
# -*- coding: Latin-1 -*-

from Tkinter import *

def nyomtat():
    ch1 = e.get()                 # az Entry widget egy utf8 stringet ad vissz
    ch2 = ch1.encode("Latin-1")   # utf8 -> Latin-1 konverzió
    print ch2

f = Tk()
e = Entry(f)
e.pack()
Button(f, text ="kiírni", command = nyomtat).pack()
f.mainloop()
```

Próbáljuk ki ezt a kis scriptet úgy, hogy az adatbeviteli mezőbe ékezetes karaktereket tartalmazó stringet írunk be.

Tegyünk még egy próbát úgy, hogy a « print ch2 » utasítást kicseréljük a « print ch1 » -gyel. Vonjuk le a következtetést !

14.4.3 « Scrolled Text »

Ez a megawidget úgy terjeszti ki a standard **Text** widget lehetőségeit, hogy egy keretet, egy címkét (a címet) és görgetősávokat kapcsol hozzá.

A következő script bemutatja, hogy a « ScrolledText » mega-widget alapvetően szövegek kiíratására való, de formázhatjuk a szövegeket és képeket építhetünk beléjük.

A kiírt elemeket (szövegeket vagy képeket) « klikkelhetővé » tehetjük és arra használhatjuk, hogy mindenféle mechanizmust elindítsunk.

A fenti ábrát létrehozó alkalmazásban például a « Jean de la Fontaine » névre kattintva a szöveg automatikusan addig gördül (*scrolling*), amíg láthatóvá nem válik ebben a widgetben egy olyan rész, ami ezt a szerzőt írja le. (Lásd a következő oldalon a megfelelő scriptet).

Más funciók is vannak, de csak a legalapvetőbbeket fogom itt bemutatni. Aki ezekről többet akar megtudni, az nézze meg a *Pmw*-t kísérő demókat és példákat.

A kiírt szöveg kezelése : két kiegészítő eszköz- az ***index*** és a ***tag-ek*** - segítségével lehet hozzáférni a kezelt szöveg bármelyik részéhez :

- A szöveg minden egyes kiírt karakterére egy index hivatkozik, aminek két egymással ponttal összekapcsolt numerikus értékből képzett stringnek kell lenni (pl: " 5.2 "). Ez a két érték jelenti azt a sorszámot és oszlopszámot, ahol a karakter van.
- A szöveg bármely részét összekapcsolhatjuk egy vagy több ***tag***-gel, ami(k)nek a nevét és tulajdonságait szabadon választjuk meg. Ezek teszik lehetővé a betűtípus, az elő- és háttérszín, az asszociált események, stb. definiálását.

Megjegyzés : Az alábbi script jobb megértéséhez tegyük fel, hogy a kezelendő szöveg egy « CorbRenard.txt » nevű fileban van.

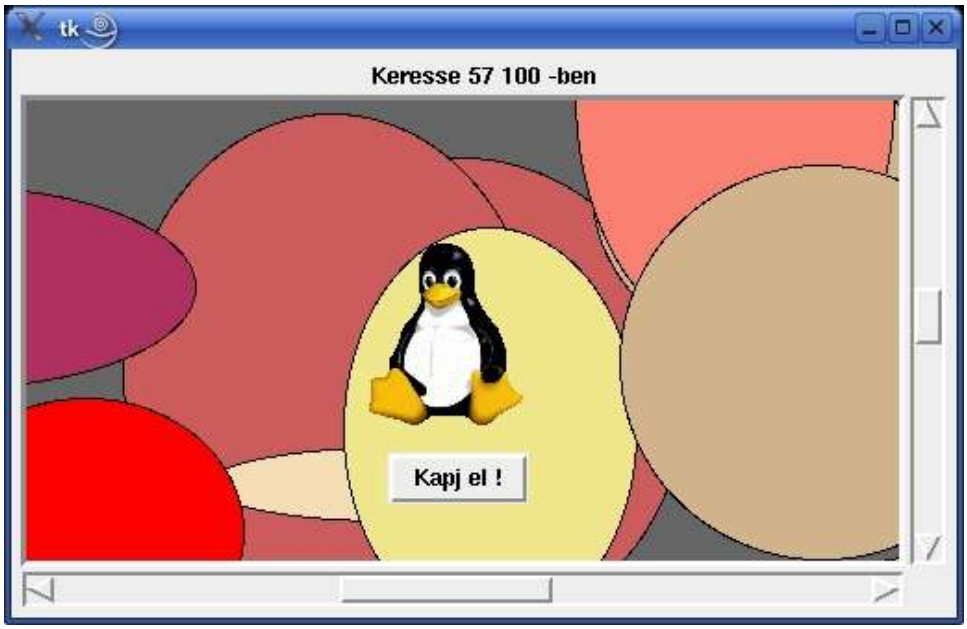
```
1.   from Tkinter import *
2.   import Pmw
3.
4.   def action(event=None):
5.       """a szöveg görgetése a <target> tag-ig"""
6.       index = st.tag_nextrange('target', '0.0', END)
7.       st.see(index[0])
8.
9.   # Egy ScrolledText widgetet tartalmazó ablak létrehozása  :
10.  fen = Pmw.initialise()
11.  st = Pmw.ScrolledText(fen,
12.                        labelpos =N,
13.                        label_text ="A ScrolledText widget demója",
14.                        label_font ='Times 14 bold italic',
15.                        label_fg = 'navy', label_pady =5,
16.                        text_font='Helvetica 11 normal', text_bg ='ivory',
17.                        text_padx =10, text_pady =10, text_wrap ='none',
18.                        borderframe =1,
19.                        borderframe_borderwidth =3,
20.                        borderframe_relief =SOLID,
21.                        usehullsize =1,
22.                        hull_width =370, hull_height =240)
23.  st.pack(expand =YES, fill =BOTH, padx =8, pady =8)
24.
25.  # Tag-ek definíciója, eseménykezelő kapcsolása az egérkattintáshoz :
26.  st.tag_configure('title_', foreground ='brown', font ='Helvetica 11 bold italic')
27.  st.tag_configure('link', foreground ='blue', font ='Helvetica 11 bold')
28.  st.tag_configure('target', foreground ='forest green', font ='Times 11 bold')
29.  st.tag_bind('link', '<Button-1>', action)
30.
31.  title_ ="""A róka és a holló
32.  írta Jean de la Fontaine, francia szerző
33.  \n"""
34.  author ="""
35.  Jean de la Fontaine
36.  francia író (1621-1695)
37.  Verses meséi és mindenek előtt,
38.  1668-tól 1694-ig publikált állattörténetei
39.  tették híressé."""
40.
41.  # A Text widget kitöltése (2 technika) :
42.  st.importfile('CorbRenard.txt')
43.  st.insert('0.0', title_, 'title_')
44.  st.insert(END, author, 'target')
45.  # Kép beszúrása :
46.  photo =PhotoImage(file= 'penguin.gif')
47.  st.image_create('6.14', image =photo)
48.  # tag dinamikus létrehozása :
49.  st.tag_add('link', '2.4', '2.23')
50.
51.  fen.mainloop()
```

Magyarázatok :

- 4. - 7. sor : Ez egy eseménykezelő. Hívására a szerző nevére történő kattitáskor kerül sor (27. és 29. sorok). A 6. sorban a widget **tag_nextrange()** metódusát használjuk a « target » ***tag***-hez asszociált szövegrész ***indexeinek*** a megkeresésére. Az indexek keresése a 2. és 3. argumentummal megadott tartományra korlátozódik (példánkban a szöveg elejétől a végéig keresünk). A **tag_nextrange()** metódus visszatérési értékként egy két indexből álló listát ad meg (a « target » tag-gal asszociált szövegrész első és utolsó karakterének indexét). A 7. sorban az indexek az egyikét (az elsőt) használjuk a **see()** metódus aktiválására. Ez a metódus a szöveg automatikus görgetését (*scrolling*) idézi elő úgy, hogy a megadott indexnek megfelelő karakter válik láthatóvá a widgetben (általában az őt követő néhány karakterrel együtt).
- 9. - 23. sorok : Egyetlen egy widget kiírására szánt ablak létrehozása. A widget-objektumot létrehozó kódba belevettem néhány opciót azzal a céllal, hogy bemutassam a számos konfigurációs lehetőség egy részét.
- 12. sor : A **labelpos** opció meghatározza a címke (cím) elhelyezkedését a szövegablakhoz képpest. Az égtájak jelölésére használt betűket fogadja el értékként (N, S, E, W, NE, NW, SE, SW). Ha nem akarunk címkét kiírni, akkor ezt az opciót nem kell használni.
- 13. - 15. sorok : A címke nem más, mint egy - az összetett **ScrolledText** widgetbe beépített - standard **Label** widget. A következő szintaxissal minden konfigurációs opciójához hozzáférhetünk : az elő- és háttérszín, a betűtípus, a méret, sőt a widget körül fenntartott térköz (**pady** opció) egyszerű definiálásához a **label_** prefixet annak az opciónak a nevéhez kell kapcsolni, amit aktiválni akarunk.
- 16. - 17. sorok : A **ScrolledText**-be beépített **Text** widget konfigurációs opcióihoz a címkére leírthoz hasonló technikát használva férhetünk hozzá. Most a **text_** prefixet kell az opciók nevéhez kapcsolni.
- 18. - 20. sorok : A **Text** widget köré egy keretet (egy **Frame** widgetet) akarok tenni. A **borderframe = 1** opcióval jeleníthetjük meg. A konfigurációs opcióihoz a **label_** és a **text_**-re leírtakhoz hasonlóan férhetünk hozzá.
- 21. - 22. sorok : Ezekkel az opciókkal globálisan rögzíthetjük a widget méreteit. Egy másik lehetőség az lenne, hogy inkább a **Text** komponens méreteit definiálnánk (például a **text_width** és **text_height** opciókkal). Ebben az esetben azt kockáztatnánk, hogy a widget globális méretei a tartalma függvényében változnának (a gördítősávok automatikus megjelenése/eltűnése). Megjegyzés *: a **hull** szó a globális konténert, azaz magát a mega-widgetet jelöli.*
- 23. sor : A **pack()** metódus **expand = YES** és **fill = BOTH** opciói azt jelentik, hogy az illető widgetet vízszintesen és függőlegesen át lehet méretezni.
- 26 - 29. sorok : Ezek a sorok definiálják a « title_ », « link » és « target » ***tag***-eket és azoknak a szövegeknek a formázását, amik hozzájuk lesznek asszociálva. A 29. sor ráadásul előírja, hogy a « link » tag-hez asszociált szöveg klikkelhető lesz és megadja a megfelelő eseménykezelőt.
- 42. sor : Szöveg importálása egy fileból. Megjegyzés : Második argumentumként egy indexet megadva meghatározható a pontos hely, ahová be kell szúrni a szöveget.
- 43. -44. sorok : Ezek az utasítások úgy szúrják be a szövegtöredékeket (a már létező szöveg elejéhez és végéhez), hogy mindegyikükhöz egy tag-et asszociálnak.
- 49. sor : A tag-ek hozzákapcsolása a szövegekhez dinamikus. Bármelyik pillanatban új asszociációt aktiválhatunk (ahogy a « link » tag-nak egy már létező szövegrészhez való kapcsolásával tesszük) Megjegyzés : a **tag_delete()** metódust használjuk a tag lekapcsolására.().

14.4.4 « Scrolled Canvas »

A következő script bemutatja hogyan lehet kihasználni a **ScrolledCanvas** (görgetett vászon) mega-widget-et, ami a standard **Canvas** (vászon) widget lehetőségeit terjeszti ki, görgetősávokat, egy címkét és egy keretet kapcsolva hozzá. A példa egy játék, melyben a felhasználónak rá kell kattintani egy állandóan mozgó gombra. (*Megjegyzés : ha nehéz elkapni a gombot, kezdje először az ablak megnyújtásával*)



A **Canvas** widget nagyon sokoldalú : lehetővé teszi, rajzok, *bitmap* képek, szövegrészek, sőt más widgetek kombinálását egy tökéletesen széthúzható térben. Ha valamilyen grafikus játékot akarunk írni, akkor nyilvánvaló, hogy ez az a widget, amit elsődlegesen meg kell tanulni uralni.

A jegyzetben ebben a tárgykörben közölt ismeretek szükégszerűen nagyon hiányosak. A célom csak az, hogy segítsek megérteni néhény alapfogalmat, hogy aztán az olvasó képes legyen a speciális referencia művekben információt keresni.

Az alkalmazásunk a **Pmw.ScrolledCanvas()** mega-widget osztályából származtatott új **FenPrinc()** osztályként jelenik meg. Egy görgető sávokkal ellátott nagy vásznat tartalmaz, amiben 80 színes, véletlenszerű elhelyezkedésű és méretű ellipszis van.

Egy kis bitmap formátumú képet teszünk rá, amit mindenek előtt arra szánok, hogy az emlékezetünkbe idézzem, hogyan lehet ezt a típusú erőforrást kezelni.

Végül egy valódi widget-et, jelen esetben egy gombot installálunk rá, de a kidolgozott technikát bármely más típusú widgetre alkalmazhatnánk, akár olyan nagy összetett widgetekre is, mint amilyeneket az előzőekben fejlesztettünk. Ez a nagy rugalmasság a komplex alkalmazások fejlesztésében az egyik fő nyereség, amit az objektum orientált programozás nyújt.

Az első pillanattól kezdve, hogy rákattintottunk, a gomb megelevenedik. A következő scriptanalízisben figyeljen az olvasó azokra a metódusokra, amiket egy létező objektum tulajdonságainak módosítására használok.

```
1.   from Tkinter import *
2.   import Pmw
3.   from random import randrange
4.
5.   Pmw.initialise()
6.   colo =['sienna','maroon','brown','pink','tan','wheat','gold','orange','plum',
7.          'red','khaki','indian red','thistle','firebrick','salmon','coral']
8.
9.   class MainWind(Pmw.ScrolledCanvas):
10.      """Főablak : nyújtható vászon görgetősávokkal"""
11.      def __init__(self):
12.          Pmw.ScrolledCanvas.__init__(self,
13.                  usehullsize =1, hull_width =500, hull_height =300,
14.                  canvas_bg ='grey40', canvasmargin =10,
15.                  labelpos =N, label_text ='Kapd el a gombot !',
16.                  borderframe =1,
17.                  borderframe_borderwidth =3)
18.          # Az alábbi opciókat az inicializálás után meg kell adni :
19.          self.configure(vscrollmode ='dynamic', hscrollmode ='dynamic')
20.          self.pack(padx =5, pady =5, expand =YES, fill =BOTH)
21.
22.          self.can = self.interior()        # hozzáférés a vászon-komponenshez
23.          # Dekoráció : véletlen ellipszisek rajzolása :
24.          for r in range(80):
25.              x1, y1 = randrange(-800,800), randrange(-800,800)
26.              x2, y2 = x1 + randrange(40,300), y1 + randrange(40,300)
27.              colour = colo[randrange(0,16)]
28.              self.can.create_oval(x1, y1, x2, y2, fill=colour, outline='black')
29.          # Kisméretű GIF kép hozzáadása :
30.          self.img = PhotoImage(file ='linux2.gif')
31.          self.can.create_image(50, 20, image =self.img)
32.          # Annak a gombnak a kirajzolása, amit el kell kapni :
33.          self.x, self.y = 50, 100
34.          self.but = Button(self.can, text ="Start", command =self.start)
35.          self.fb = self.can.create_window(self.x, self.y, window =self.but)
36.          self.resizescrollregion()
37.
38.      def anim(self):
39.          if self.run ==0:
40.              return
41.          self.x += randrange(-60, 61)
42.          self.y += randrange(-60, 61)
43.          self.can.coords(self.fb, self.x, self.y)
44.          self.configure(label_text = 'Keress en %s %s' % (self.x, self.y))
45.          self.resizescrollregion()
46.          self.after(250, self.anim)
47.
48.      def stop(self):
49.          self.run =0
50.          self.but.configure(text ="Restart", command =self.start)
51.
52.      def start(self):
53.          self.but.configure(text ="Kapj el !", command =self.stop)
54.          self.run =1
55.          self.anim()
56.
57.  ##### Főprogram ##############
58.
59.  if __name__ == '__main__':
60.      MainWind().mainloop()
```

Magyarázatok :

- 6. sor : A *Tkinter* ezeket a színneveket mind elfogadja. Nyilván helyettesíthetnénk őket hexadecimális értékekkel, ahogy a 201. oldalon magyaráztam.
- 12. - 17. sorok : Ezek az opciók nagyon hasonlítanak a ***ScrolledText*** widgetnél leírt opciókra. A megawidget egy **Frame,** egy **Label,** egy **Canvas** és két **Scrollbar** komponenst egyesít. A komponensek konfigurációs opcióihoz olyan szintaxissal férünk hozzá, ami a komponens és az opció nevét egy « _ » karakterrel kapcsolja össze.
- 19. sor : Ezek az opciók definiálják a görgető sávok megjelenési módját. « Statikus » módban a görgető sávok mindíg jelen vannak. « Dinamikus » módban eltűnnek, ha a vászon méretei az ablak méreteinél kisebbé válnak.
- 22. sor : Az **interior()** metódus a **ScrolledCanvas** mega-widgetbe integrált **Canvas** komponens hivatkozását adja visszatérési értékként. A következő sorok (23.-35. sorok) elemeket helyeznek erre a vászonra : rajzokat, egy képet és egy gombot.
- 25. - 27. sorok : A **randrange()** függvénnyel állíthatók elő egy megadott intervallumba eső véletlen egész számok. (Lásd a 142.lapon lévő magyarázatot.).
- 35. sor : A **Canvas** widget **create_window()** metódusával szúrhatunk be bármely más widget-et (egy összetett widget-et is). A beszúrandó widget-et azonban előzőleg mint a vászon vagy a vászon master-ablakának slave-jét kell definiálni. A **create_window()** metódus három argumentumot vár : annak a pontnak az X és Y koordinátáit, ahova be akarjuk szúrni a widget-et és a beszúrandó widget hivatkozását.
- 36. sor : A **resizescrollregion()** metódus újra beállítja a görgető sávok helyzetét úgy, hogy azok megfeleljenek az aktuálisan megjelenített vászonrésznek.
- 38. - 46. sorok : Ez a metódus a gomb animálására szolgál. Miután a gombot az előző pozíciójától valamilyen távolságra véletlenszerűen áthelyezte, 250 msec után a metódus újra hívja önmagát. Ez hurkolás mindaddig megállás nélkül folyik, amíg a self.run változó értéke nem nulla.
- 48. - 55. sorok : Ez a két eseménykezelő felváltva van a gombhoz kapcsolva. Nyilvánvalóan az animáció indítására és leállítására valók.

14.4.5 Eszköztárak buborék helppel - lambda kifejezések

Sok programnak van egy vagy több – kis piktogramokkal (ikonokkal) ellátott gombokból álló - eszköztára (*toolbar*). Az eszköztár(ak)ban úgy kínálhatunk fel nagyszámú specializált parancsot a felhasználónak, hogy azok nem foglalnak el nagy helyet a képernyőn (azt mondják: egy kis rajz többet ér, mint a hosszú beszéd).

Az ikonok jelentése azonban nem mindíg nyilvánvaló, különösen az új felhasználóknak. Ajánlatos ezért kiegészíteni az eszköztárakat egy buborék helprendszerrel (*tool tips*), ami rövid magyarázó üzenetekből áll, melyek akkor jelennek meg, amikor az egérkurzor az illető gomb fölé kerül.

A következő alkalmazás egy eszköztárból és egy vászonból áll. Amikor a felhasználó az eszköztár egyik gombjára kattint, akkor a script a gombon lévő ikont a vászon véletlenszerűen kiválasztott helyére másolja :

Példánkban mindegyik gombot egy mélyedés veszi körül. A gombokat létrehozó utasításban a **relief** (domborzat) és a **bd** (border = szél) opciók megfelelő megválasztásával könnyen kaphatunk más kinézetű gombokat. A **relief = FLAT** és **bd =0** választással « sík » gombokat kapunk.

A help buborékok elhelyezése gyerekjáték. Az egész alkalmazás számára elég egyetlen **Pmw.Balloon** objektumot létrehozni. Utána ennek a **Pmw.Balloon** objektumnak a **bind()** metódusát annyiszor hívva ahányszor szükséges, mindegyik widgethez, amelyikhez help buborékot szeretnénk kapcsolni, szöveget asszociálunk.

```
1.  from Tkinter import *
2.  import Pmw
3.  from random import randrange
4.
5.  # az ikonokat tartalmazó file-ok nevei (GIF formátumú ):
6.  images =('floppy_2','papi2','pion_1','pion_2','help_4')
7.  textes =('mentés','pillangó','játékos 1','játékos 2','Help')
8.
9.  class Application(Frame):
10.     def __init__(self):
11.         Frame.__init__(self)
12.         # Egy <buborék help> objektum létrehozása (egy elegendő) :
13.         tip = Pmw.Balloon(self)
```

```
14.          # Eszköztár létrehozása (ez egy egyszerű keret) :
15.          toolbar = Frame(self, bd =1)
16.          toolbar.pack(expand =YES, fill =X)
17.          # A létrehozandó gombok száma :
18.          nBut = len(images)
19.          # A gombok ikonjainak persistens változókban kell lenni.
20.          # Egy lista megteszi :
21.          self.photoI =[None]*nBut
22.
23.          for b in range(nBut):
24.              # Ikon létrehozása (PhotoImage Tkinter objektum) :
25.              self.photoI[b] =PhotoImage(file = images[b] +'.gif')
26.
27.              # Gomb létrehozása.:
28.              # Egy "lambda" kifejezést használunk arra hogy a hívott
29.              # metódusnak parancsként adjunk át egy argumentumot :
30.              but = Button(toolbar, image =self.photoI[b], relief =GROOVE,
31.                           command = lambda arg =b: self.action(arg))
32.              but.pack(side =LEFT)
33.
34.              # egy gomb összekapcsolása egy helpszöveggel :
35.              tip.bind(but, textes[b])
36.
37.          self.ca = Canvas(self, width =400, height =200, bg ='orange')
38.          self.ca.pack()
39.          self.pack()
40.
41.      def action(self, b):
42.          "a gomb ikonja a vászonra van másolva"
43.          x, y = randrange(25,375), randrange(25,175)
44.          self.ca.create_image(x, y, image =self.photoI[b])
45.
46.  Application().mainloop()
```

Metaprogramozás. Lambda kifejezések :

Az általános szabály : mindegyik gombhoz egy ***parancsot*** kapcsolunk, ami egy metódus vagy egy speciális függvény. Ez felelős a feladat elvégzéséért, amikor a gomb aktiválva van. Viszont ebben az alkalmazásban minden gombnak majdnem ugyanazt kell csinálni (egy rajzot kell átmásolni a vászonra), az egyetlen különbség közöttük az illető rajz.

Kódunk egyszerűsítése érdekében mindegyik gombunk **command** opcióját ugyanazzal a metódussal (ez az **action()** metódus) szeretnénk összekapcsolni, de minden alkalommal úgy akarjuk átadni a használt speciális gomb referenciáját, hogy mindegyik gomb esetén különböző lehessen a végrehajtott akció.

Felmerül egy nehézség : a **Button** *widget* **command** opciója csak egy ***értéket*** vagy ***kifejezést*** fogad el, ***utasítást*** nem. Tehát megadható neki egy függvényhivatkozás, de nem hívhatja a függvényt esetleges argumentumok átadásával (ez az oka amiért a függvény nevét zárójelek nélkül adjuk meg).

Ez a nehézség kétféleképpen oldható meg:

- ***Dinamikus*** jellegéből adódóan a Python elfogadja, hogy ***egy program módosítani tudja önmagát*** például úgy, hogy a végrehajtása során új függvényeket definiál (ezt jelenti a ***metaprogramozás*** fogalma).

  Tehát futás közben definiálható egy paramétereket használó függvény úgy, hogy ***a paraméterek mindegyikének megadunk egy alapértelmezett értéket***, majd ezt a függvényt hívjuk argumentumok nélkül ott, ahol az argumentumok megadása nem megengedett. Mivel a függvény a programvégrehajtás során van definiálva, ezért az alapértelmezett értékek lehetnek

változóértékek. A művelet eredménye egy valódi argumentumátadás.

E technika illusztrációjaként helyettesítsük a script 27. - 31. sorait a következőkkel :

```
# Gomb létrehozása.:
# Futás közben definiálunk egy paraméteres függvényt. A paraméter
# alapértelmezett értéke az átadandó argumentum.
# Ez a függvény hív egy argumentumot igénylő metódust :

def act(arg = b):
    self.action(arg)

# A gombhoz kapcsolt parancs hívja a fenti függvényt :
bou = Button(toolbar, image =self.photoI[b], relief =GROOVE,
                  command = act)
```

- Az előzőeket egy ***lambda*** kifejezés hívásával egyszerűsíthetjük. Ez a foglalt Python szó egy olyan ***kifejezést*** jelöl, ami egy függvényobjektumot ad vissza a **def** utasításhoz hasonlóan. A különbség az, hogy a lambda egy kifejezés és nem utasítás, ezért interfaceként tudjuk használni egy függvény (argumentumátadással) történő hívására ott, ahol az a szokásos módon nem lehetséges. Jegyezzük meg, hogy az ilyen függvény anonym (nincs neve).

  Például a :

  ```
  lambda ar1=b, ar2=c : akarmi(ar1,ar2)
  ```

  utasítás egy anonym függvény hivatkozását adja visszatérési értékül. Ez a függvény a **b** és **c** argumentumokkal - azokat a függvényparaméterek definíciójában alapértelmezett értékekként használva - az **akarmi()** függvényt fogja hívni.

  Ez a technika végsősoron ugyanazt az elvet használja mint az előző. Az előnye, hogy tömörebb, ezért használom a scriptben. Viszont kicsit nehezebb megérteni :

  ```
  command = lambda arg =b: self.action(arg)
  ```

  Ebben az utasításrészben a gombhoz rendelt parancs egy anonym függvényre hivatkozik. A függvény **arg** paraméterének az alapértelmezett értéke a **b** argumentum értéke.
  Amikor a parancs argumentum nélkül hívja, ez az anonym függvény mégis használni tudja a paraméterét (az alapértelmezett értékkel) a **self.action()** célfüggvény hívására és így egy valódi argumentum átadást érünk el.

- Nem részletezem a lambda kifejezések kérdését, mert ez meghaladja ennek a bevezető műnek a kereteit. Ha többet szerene tudni róluk, akkor nézze meg a bibliográfiában megadott valamelyik referencia művet.

14.5 Ablakok menükkel

Különböző típusú legördülő menükkel ellátott alkalmazásablak konstrukcióját fogom most leírni. Mindegyik menüt le lehet választani a főalkalmazásról, hogy független ablakokká váljon, mint az alábbi illusztráción.

Ez a kicsit hosszabb gyakorlat áttekintésül is fog szolgálni. Szakaszosan fogjuk kivitelezni az ***inkrementális fejlesztés***nek nevezett programozási stratégia alkalmazásával.

Amint már az előzőekben magyaráztam [55], ez a módszer abból áll, hogy a programírást csak egy néhány sorból álló vázzal kezdjük, de ez már működik. Gondosan teszteljük, hogy kiküszöböljük az esetleges hibákat. Amikor korrekten működik, akkor hozzáadunk egy kiegészítő funkciót. Addig teszteljük ezt a kiegészítést, amíg teljesen kielégítő nem lesz, majd hozzáadunk a vázhoz egy újabb funkciót és így tovább ...

Ez nem jelenti azt, hogy a projekt előzetes, komoly elemzése nélkül ***rögtön neki lehet a kezdeni programozásnak.*** A projektet legalább nagy vonalakban megfelelő módon le kell írni egy világosan megfogalmazott ***feladat meghatározásban***.

A kidolgozás során a kód megfelelő kommentezésére is szükség van. *Törekedni kell a jól megfogalmazott kommentekre, nemcsak azért, hogy a kódunkat könnyen lehessen olvasni (és később másoknak vagy magunknak könnyebb legyen karbantartani), hanem azért is, hogy kényszerítsük magunkat annak a kifejtésére, amit szeretnénk, hogy a gép valóban csináljon. (Vö. Szemantikai hibák, 16. oldal)*

Feladat meghatározás :

Az alkalmazásunknak lesz egy menüsora és egy vászna. A menü különböző rovatai és opciói csak arra szolgálnak, hogy szövegrészleteket jelenítenek meg a vásznon vagy a dekoráció részleteit módosítják és mindenek előtt változatos példák lesznek. Ezeknek az a rendeltetésük, hogy vázlatát adják annak a sokféle lehetőségnek, amit ez a widget-típus kínál, ami nélkülözhetetlen tartozéka minden modern alkalmazásnak.

Az is kívánalom, hogy a gyakorlat során előállított kód jól struktúrált legyen. Ennek érdekében két osztályt fogok használni : egyik a főalkalmazásé, a másik a menüsoré. Azért akarok így eljárni, hogy megvilágítsam egy több interaktív objektum osztályait magába foglaló alkalmazástípus konstrukcióját.

55 Lásd : Hibakeresés és kísérletezés (16.oldal)

14.5.1 A program első váza :

Egy program vázának a megalkotásakor arra kell törekedni, hogy a lehető legkorábban jelenítsük meg a teljes struktúrát a végleges alkalmazást alkotó fő blokkok közötti relációkkal együtt. Erre törekszünk az alábbi példában :

```
1.  from Tkinter import *
2.
3.  class MenuBar(Frame):
4.      """Legördülő menük sora"""
5.      def __init__(self, boss =None):
6.          Frame.__init__(self, borderwidth =2)
7.
8.          ##### <File> menu #####
9.          fileMenu = Menubutton(self, text ='File')
10.         fileMenu.pack(side =LEFT)
11.         # A "legördülő" rész :
12.         me1 = Menu(fileMenu)
13.         me1.add_command(label ='Törlés', underline =0,
14.                         command = boss.erase)
15.         me1.add_command(label ='Befejezés', underline =0,
16.                         command = boss.quit)
17.         # A menü integrálása :
18.         fileMenu.configure(menu = me1)
19.
20. class Application(Frame):
21.     """Főalkalmazás"""
22.     def __init__(self, boss =None):
23.         Frame.__init__(self)
24.         self.master.title('Ablak menükkel')
25.         mBar = MenuBar(self)
26.         mBar.pack()
27.         self.can = Canvas(self, bg='light grey', height=190,
28.                           width=250, borderwidth =2)
29.         self.can.pack()
30.         self.pack()
31.
32.     def erase(self):
33.         self.can.delete(ALL)
34.
35. if __name__ == '__main__':
36.     app = Application()
37.     app.mainloop()
```

Kódoljuk a fenti sorokat és ellenőrizzük a végrehajtást. Egy ablakot kell kapnunk egy világosszürke vászonnal, ami fölött egy menüsor van. Ebben a stádiumban a menüsornak csak egy « File » rovata van.

Kattintsunk a « File » rovatra a megfelelő menü megjelenítéséhez : a « Törlés » opció még nem működik (a vászon tartalmát fogja törölni), de a « Befejezés » opciónak már lehetővé kell tenni az alkalmazás megfelelő lezárását.

Mint a *Tkinter* által kezelt valamennyi menü, úgy az általunk létrehozott menü is átalakítható « úszó » menüvé : ehhez elég rákattintani a menüfejben megjelenő szaggatott vonalra. Így egy kis szatellit ablakot kapunk, amit oda pozícionálhatunk, ahová tetszik.

A script elemzése :

A program struktúrájának már ismerősnek kell lenni. Hogy a scriptben definiált osztályok esetleg más projektekben importálással újra felhasználhatók legyenek, a főprogram teste (35. - 37. sorok) tartalmazza az : `if __name__ == '__main__':` utasítást[56].

A következő két utasítás csak egy **app** objektumot hoz létre és elindítja az objektum **mainloop()** metódusát. Ezt a két utasítást összesűríthettük volna egyetlen utasításba.

A program lényege azonban az előző osztályok-definícióiban rejlik :

A **MenuBar()** osztály tartalmazza a menüsor leírását. A script jelenlegi állapotában ez egy constructor-vázat zár magába.

- 5. sor : A **boss** paraméter a widget master ablakának referenciáját fogadja a **MenuBar()** objektum létrehozásának pillanatában. Ez a referencia fogja lehetővé tenni a master ablakhoz kapcsolt metódusok hívását a 14. és a 16. sorban.
- 6. sor : A szülőosztály constructor-ának kötelező aktiválása.
- 9. sor : A **Menubutton()** osztály egy widget-jét hozzuk létre, ami a **self** (vagyis az összetett « menüsor »-objektum, aminek az osztályát éppen most definiáljuk) egy slave-jeként van definiálva. Ahogyan a neve mutatja, ez a widgettípus egy gombhoz hasonlóan viselkedik : akkor következik be egy akció, amikor rákattintunk.
- 12. sor : Ahhoz, hogy az akció egy menü megjelenéséből álljon, még a menüt definiálni kell. A menu most a **Menu()** osztály egy új widgetje lesz, amit a 9. sorban létrehozott *Menubutton* widget « slave »-jeként definiálunk.
- 13. - 16. sorok : A **Menu()** osztály widgetjeire speciális metódusokat alkalmazhatunk, melyek mindegyike számos opciót fogad el. Az **add_command()** metódus helyezi el a « Törlés » és « Befejezés » menüpontokat a menüben. Rögtön beépítjük az **underline** opciót, ami egy ***billentyű rövidítés*** definiálására való. Ez az opció adja meg, hogy a menüpont melyik betűjének kell aláhúzva megjelenni a képernyőn. A felhasználó tudja, hogy a menüpontnak megfelelő akció aktiválásához elég lenyomni ezt a billentyűt (mintha az egérrel rákattintott volna a menüpontra).

  Egy menüpont kiválasztásakor indítandó akciót a **command** opció határozza meg. Scriptünkben mindkét hívott parancs ***a master-ablak metódusa***. A master-ablak referenciáját a **boss** paraméter adta át létrehozása pillanatában az aktuális widgetnek. Az **erase()** (töröl) metódus, amit később definiálunk, fogja törölni a vásznat. Az előre definiált **quit()** metódus idézi elő a kilépést a **mainloop()**-ból és így az alkalmazásablakhoz kapcsolt eseményfigyelő leállítását.
- 18. sor : Amikor a menüpontokat definiáltuk, utána a *Menubutton* master widgetet újra kell konfigurálnunk, hogy a « menu » opciója arra a *Menü*-re mutasson, amit megkonstruáltunk. Ezt az opciót nem tudtuk közelebbről meghatározni a *Menubutton* widget definíciójakor, mert abban a stádiumban a *Menu* még nem létezett. A *Menu* widgetet sem tudtuk elsőnek definiálni, mert azt a *Menubutton* widget « slave »-jeként kell definiálni. Három lépésben kell eljárnunk, ahogyan tettünk, és a **configure()** metódust kell hívni. (Ezt a metódust bármelyik már létező widgetre alkalmazhatjuk a widget egyik vagy a másik opciójának módosítására).

56 Lásd 174: Osztálykönyvtárakat tartalmazó modulok

Az **Application()** osztály tartalmazza a program föablakának a leírását, valamint a hozzákapcsolt eseménykezelő metódusokat.

- 20. sor : Alkalmazásunkat inkább a **Frame()** osztályból származtatjuk, aminek számos opciója van, mint a **Tk()** ős-osztályból. Így az alkalmazás teljes egészében be van zárva egy widgetbe, amit esetleg később egy nagyobb alkalmazásba lehet integrálni. Minden esetre emlékezzünk rá, hogy a *Tkinter* automatikusan létrehoz egy **Tk()** típusú master ablakot ennek a *Frame*-nek a tartalmazására.
- 23. - 24. sorok : A szülőosztály constructorának kötelező aktiválása után a **master** attribútumot, - amit a *Tkinter* minden egyes widgettel automatikusan asszociál - használjuk az alkalmazás főablakára (a master ablakra, amiről az előző bekezdésben beszéltem) való hivatkozásra és a címsor újradefiniálására.
- 25. - 29. sorok : Két slave widget létrehozása a fő *Frame*-ünk számára. A « menüsor » nyilván a másik classban definiált widget.
- 30. sor : Mint bármely más widget-et, a *Frame*-ünket pozícionálni kell.
- 32. - 33. sorok : A vászon törlésére szolgáló metódus ebben az osztályban van definiálva (mert a vászon-objektum ennek a része), de egy másik osztályban definiált slave widget *command* opciójával hívjuk. Ez a slave widget, amint azt föntebb magyaráztam, a **boss** paraméter közvetítésével veszi át a master widget-jének a referenciáját. Valamennyi referencia a pont operátoros minősített névmegadással van hierarchiába rendezve.

14.5.2 A« Zenészek » menü hozzáadása

Folytassuk programunk fejlesztését úgy, hogy a **MenuBar()** class constructorához hozzáírjuk (a 18. sor után) a következő sorokat :

```
        ##### Menu <Zeneszek> #####
        self.musi = Menubutton(self, text ='Zenészek')
        self.musi.pack(side =LEFT, padx ='3')
        # A <Zeneszek> menu legördülő része :
        me1 = Menu(self.musi)
        me1.add_command(label ='17. század', underline =1,
                        foreground ='red', background ='yellow',
                        font =('Comic Sans MS', 11),
                        command = boss.showMusi17)
        me1.add_command(label ='18. század', underline =1,
                        foreground='royal blue', background ='white',
                        font =('Comic Sans MS', 11, 'bold'),
                        command = boss.showMusi18)
        # A menü integrálása :
        self.musi.configure(menu = me1)
38.
```

... és az **Application()** class-hoz hozzáírjuk (a 33. sor után) a következő metódus definíciókat :

```
    def showMusi17(self):
        self.can.create_text(10, 10, anchor =NW, text ='H. Purcell',
                    font=('Times', 20, 'bold'), fill ='yellow')

    def showMusi18(self):
        self.can.create_text(245, 40, anchor =NE, text ="W. A. Mozart",
                    font =('Times', 20, 'italic'), fill ='dark green')
```

A sorok hozzáadása után mentsük scriptünket és hajtassuk végre.

A menüsorunk most egy kiegészítő menüt tartalmaz : a « Zenészek » menüt.

A megfelelő menü két elemet kínál föl, amiket testre szabott színnel és betűkészlettel ír ki. Ezekből a díszítő technikákból ötletet meríthetünk saját projektjeinkhez. Mértékkel alkalmazzuk őket !

Nyivánvalóan azért, hogy ne nehezítsem meg a gyakorlatot, a menüpontokhoz kapcsolt parancsok egyszerűek : rövid szövegeket írunk ki a vászonra.

A script elemzése

Csak néhány újdonság van ezekben a sorokban a meghatározott betűkészletek (**font** opció), a kiírt szövegek elő- és háttérszínének (**foreground** és **background**) használatára vonatkozóan.

Ismét figyeljük meg az **underline** opció használatát, ami a billentyű rövidítéseknek megfelelő karaktert adja meg (ne felejtsük el, hogy a stringek karaktereinek számozása nullától kezdődik), és különösen azt, hogy ezeknek a widgeteknek a **command** opciója a **boss** attribútumban tárolt referencia segítségével fér hozzá más osztály metódusaihoz.

A vászon **create_text()** metódusát két numerikus argumentummal kell használni, amik a vásznon egy pont X és Y koordinátái. Az argumentumként megadott szöveg ehhez a ponthoz képpest lesz pozícionálva az **anchor** opció választott értékétől függően. Ez az érték határozza meg, hogyan kell a vásznon kiválasztott ponthoz « lehorgonyozni » a szöveget : a középpontjánál, a balfelső sarkánál, stb. fogva. Az alkalmazott szintaxis az égtájak megadásával analóg (NW=balfelső sarok, SE=jobbalsó sarok, CENTRE=közepe, stb.).

14.5.3 A « Festők » menü hozzáadása:

Ez az új menü az előzőhöz nagyon hasonló módon van megszerkesztve, viszont hozzáadtunk egy kiegészítő funkciót : a « kaszkád » menüket. Írjuk a **MenuBar()** class constructorához a következő sorokat :

```
##### <Festők> menu #####
self.pain = Menubutton(self, text ='Festők')
self.pain.pack(side =LEFT, padx='3')
# A "legördülő" rész :
me1 = Menu(self.pain)
me1.add_command(label ='klasszikusok', state=DISABLED)
me1.add_command(label ='romantikusok', underline =0,
                command = boss.showRomanti)
# Almenü az impresszionista festőknek :
me2 = Menu(me1)
me2.add_command(label ='Claude Monet', underline =7,
                command = boss.tabMonet)
me2.add_command(label ='Auguste Renoir', underline =8,
                command = boss.tabRenoir)
me2.add_command(label ='Edgar Degas', underline =6,
                command = boss.tabDegas)
# Az almenü integrálása :
me1.add_cascade(label ='impresszionisták', underline=0, menu =me2)
# A menü integrálása :
self.pain.configure(menu =me1)
```

... és az **Application**() class-ba a következő definíciókat :

```
def showRomanti(self):
    self.can.create_text(245, 70, anchor =NE, text = "E. Delacroix",
                font =('Times', 20, 'bold italic'), fill ='blue')

def tabMonet(self):
    self.can.create_text(10, 100, anchor =NW, text = 'Nymphéas à Giverny',
                font =('Technical', 20), fill ='red')

def tabRenoir(self):
    self.can.create_text(10, 130, anchor =NW,
                text = 'Le moulin de la galette',
                font =('Dom Casual BT', 20), fill ='maroon')

def tabDegas(self):
    self.can.create_text(10, 160, anchor =NW, text = 'Danseuses au repos',
                font =('President', 20), fill ='purple')
```

A script elemzése:

Az almenük láncolásával könnyen létrehozhatók tetszőleges mélységű kaszkád menük (ennek ellenére nem ajánlott az 5 szintnél mélyebb egymásbaágyazás : a felhasználók elvesznek benne).

Egy almenü az előző menüszint « slave » menüjeként van definiálva (példánkban **me2** az **me1** « slave » menüjeként van definiálva). A beillesztést aztán az **add_cascade**() metódus biztosítja.

Az egyik menüpont inaktivvá van téve (**state** = DISABLED opció). A következő példa meg fogja mutatni, hogy hogyan lehet programból aktiválni, illetve inaktiválni menüpontokat.

14.5.4 Az « Opciók » menü beillesztése :

Ennek a menünek a definíciója egy kicsit komplikáltabb, mert *Tkinter* belső változók használatát fogjuk beleépíteni.

Viszont ennek a menünek a funkcionalitásai jóval kifinomultabbak : a hozzáadott opciók lehetővé teszik a « Zenészek » és « Festők » menüpontok szándékos aktiválását és inaktiválását, és magának a menüsornak a kinézetét is módosíthatjuk.

Írjuk a **MenuBar()** class constructorába a következő sorokat :

```
##### Menu <Options> #####
optMenu = Menubutton(self, text ='Opciók')
optMenu.pack(side =LEFT, padx ='3')
# Tkinter változók :
self.relief = IntVar()
self.actPain = IntVar()
self.actMusi = IntVar()
# A menu "legördülő része" :
self.mo = Menu(optMenu)
self.mo.add_command(label = 'Aktiválás :', foreground ='blue')
self.mo.add_checkbutton(label ='zenészek',
          command = self.choiceActive, variable =self.actMusi)
self.mo.add_checkbutton(label ='festők',
          command = self.choiceActive, variable =self.actPain)
self.mo.add_separator()
self.mo.add_command(label = 'Domborzat :', foreground ='blue')
for (v, lab) in [(0,'nincs'), (1,'kiemelkedő'), (2,'besüllyedő'),
                 (3,'árok'), (4,'gerinc'), (5,'keret')]:
    self.mo.add_radiobutton(label =lab, variable =self.relief,
                            value =v, command =self.reliefBarre)
# A menü integrálása :
optMenu.configure(menu = self.mo)
```

... ... valamint a következő metódusok definícióit (még mindíg a **MenuBar()** class-ba) :

```
def reliefBarre(self):
    choix = self.relief.get()
    self.configure(relief =[FLAT,RAISED,SUNKEN,GROOVE,RIDGE,SOLID][choix])

def choiceActive(self):
    p = self.actPein.get()
    m = self.actMusi.get()
    self.pein.configure(state =[DISABLED, NORMAL][p])
    self.musi.configure(state =[DISABLED, NORMAL][m])
```

A script elemzése

a) Menü « checkbox »-okkal

Az új legördülő menünknek két része van. Hogy ezt nyilvánvalóvá tegyem, beszúrtam egy elválasztó vonalat, valamint két « hamis menüpontot » (« Aktiválás: » és « Domborzat : »), amik csak címekként szolgálnak. Hogy a felhasználó ne keverje őket össze az igazi parancsokkal, ezért színesen íratom ki őket.

Az első rész menüpontjai checkbox-okkal vannak ellátva. Amikor a felhasználó rákattint az egérrel az egyik, vagy a másik menüpontra, akkor a megfelelő opciók aktiválva vagy inaktiválva vannak és ezeket az « aktív/inaktív » állapotokat jeleníti meg egy x. Az ilyen típusú menük elhelyezésére szolgáló utasítások elég explicitek. A menüpontokat « chekbutton » típusú widgetekként jelenítik meg :

```
self.mo.add_checkbutton(label = 'zenészek', command = choixActifs,
                        variable = mbu.me1.music)
```

Fontos annak a megértése, hogy ennek a widgettípusnak szükségszerűen van egy belső változója, ami a widget « actív / inactív » állapotának tárolására szolgál. Ez nem lehet egy közönséges Python változó, mert a *Tkinter* könyvtár osztályai más nyelven vannak megírva. Következésként egy ilyen belső változóhoz csak egy interfacen keresztül férhetünk hozzá. Ez a « *Tkinter változó* »-nak nevezett interface valójában egy objektum, amit egy speciális ***osztályból*** hozunk létre, ami ugyanúgy része a *Tkinter* modulnak, mint a widget-osztályok. Ezeknek az « objektum-változóknak » viszonylag egyszerű a használata :

- Az **IntVar()** osztály az egész típusú változókkal ekvivalens objektumok létrehozását teszi lehetővé. Tehát egy vagy több ilyen objektum-változónak a létrehozásával kezdjük, amiket példánkban új példány-attribútumokként tárolunk :
  ```
  self.actMusi =IntVar()
  ```
  Ezután az értékadás után a **self.actMusi** -ban hivatkozott objektum egy egész típusú változó ekvivalensét fogja tartalmazni speciális *Tkinter formátumban.*
- Majd a **checkbutton** objektum « variable » opcióját összekapcsoljuk az így definiált *Tkinter változóval :*
  ```
  self.mo.add_checkbutton(label ='zenészek', variable =self.actMusi)
  ```
- Azért kell így két lépésben eljárnunk, mert a *Tkinter* nem rendelhet közvetlenül értékeket Python-változókhoz. Hasonló okból a Python sem olvashatja közvetlenül egy *Tkinter változó* tartalmát. Erre ennek az objektumosztálynak egy egy speciális metódusát kell használni : a **get ()** [57] metódust :
  ```
  m = self.actMusi.get()
  ```
  Ebben az utasításban az **m** közönséges Python-változóhoz hozzárendeljük egy *Tkinter változó* tartalmát (ami maga egy meghatározott widget-hez van kapcsolva).

Az előzőek egy kicsit bonyolultnak tűnhetnek az olvasónak. Gondoljon egyszerűen arra, hogy most találkozik először két különböző programnyelv illesztésének a problémájával egy összetett projektben

57 Egy *Tkinter* változóba való ***íráshoz*** a **set()** metódust kellene használni. Példa : `self.actMusi.set (45)`

b) Menü egymást kizáró választásokkal

Az « Opciók » menü második része lehetővé teszi a felhasználónak, hogy hat lehetőség közül válassza ki a menüsor kinézetét. Magától értetődik, hogy egyszerre csak egy lehetőséget lehet aktiválni. Ennek a fajta funkcionalitásnak a hagyományos megvalósítása a « radiobutton » (rádiógomb) típusú widgetekkel történik. Ezeknek fontos jellemzője, hogy több widgetet kell ugyanahhoz a *Tkinter változóhoz* kapcsolnunk. Mindegyik rádiógombnak külön érték felel meg és ez az az érték, ami a változóhoz van rendelve, amikor a felhasználó kiválasztja a gombot.

Így a következő utasítás :

```
self.mo.add_radiobutton(label ='besüllyedő', variable =self.relief,
                        value =3, command =self.reliefBarre)
```

az «Opciók» menü egy menüpontját úgy konfigurálja, hogy az úgy viselkedik, mint egy rádiógomb.

Amikor a felhasználó kiválasztja ezt a menüpontot, akkor a **self.relief** *Tkinter változóhoz* a 3 értéket rendeli (ez a widget **variable** opciójának segítségével történik) és a **reliefBarre()** metódus hívása történik meg. Ez feladata elvégzéséhez kiolvassa a *Tkinter változóban* tárolt értéket.

Ennek a menünek a speciális összefüggésében 6 különböző lehetőséget akarunk felkínálni a felhasználónak. Tehát hat « radiobutton » -ra van szükség, amiknek a fentihez hasonló hat utasítást kódolhatnánk. Ezek csak a **value** és **label** opcióikban különböznek. Ilyen helyzetben az a jó programozási gyakorlat, hogy az opciók értékeit egy listában tároljuk, majd ezt a listát egy **for** ciklus segítségével járjuk be, hogy a widget-objektumokat egyetlen közös utasítással hozzuk létre :

```
for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                 (3,'sillon'), (4,'crête'), (5,'bordure')]:
    self.mo.add_radiobutton(label =lab, variable =self.relief,
                            value =v, command =self.reliefBarre)
```

A lista 6 tuple-ből (érték, címke) áll. A 6 iterrációs lépés mindegyikében egy új *radiobutton* elemet hozunk létre, melyek **value és label** opcióit a **v** és **lab** változók közvetítésével szedjük ki a listából.

Az olvasó gyakran fogja tapasztalni a saját projektjeiben, hogy az ilyen utasítások sorozata egy tömörebb programstruktúrával helyettesíthető (általában egy listának és egy programhuroknak a kombinációjával, mint a fenti példában).

Lassanként más technikákat is fel fog fedezni, amik a kódméretet csökkentik. A következő fejezetben még egy példát fogok adni erre. Azonban igyekezzünk észben tartani azt a fontos szabályt, hogy egy jó programnak mindenek előtt ***olvashatónak*** és ***kommentezettnek*** kell lenni.

c) Utasítás végrehajtás vezérlése listával

Most vizsgáljuk meg a **reliefBarre()** metódus definícióját :

Az első sorban a **get()** metódussal férünk hozzá egy *Tkinter változóhoz*, ami a felhasználó « Domborzat : » almenübeli választásának a számát tartalmazza.

A második sorban a **choice** változó tartalmát használjuk arra, hogy egy hatelemű listából hozzáférjünk a minket érdeklő elemhez. Például ha a **choice** tartalma 2, akkor ez a SUNKEN opció, amit a widget újrakonfigurálására fogunk használni.

A **choice** változót itt egy listaelemet kijelölő indexként használjuk. Ennek a tömör konstrukciónak a helyén a következő feltételvizsgálatot programozhattunk volna :

```
if choice ==0:
    self.configure(relief =FLAT)
elif choice ==1:
    self.configure(relief =RAISED)
elif choice ==2:
    self.configure(relief =SUNKEN)
...
etc.
```

Szigorúan funkcionális nézőpontból az eredmény ugyanaz lenne. Az olvasó azonban be fogja látni, hogy az általam választott konstrukció annál hatékonyabb, minél jobban nő a választási lehetőségek száma. Képzeljük el, hogy az egyik programunkban nagyon nagy számú elem közül kell választani : az előző konstrukcióval esetleg oda jutnánk, hogy több oldalnyi « **elif** »-et kódolnánk !

Ugyanezt a technikát használom a **choiceActive()** metódusban. Így a :

```
self.pain.configure(state =[DISABLED, NORMAL][p])
```

utasítás a **p** változó tartalmát használja indexként annak megadására, hogy a DISABLED, NORMAL állapotok melyikét kell kiválasztani a « Festők » menü újrakonfigurálásához.

Hívásakor a **choiceActive()** metódus újrakonfigurálja a menüsor « Festők » és « Zenészek » menüit, hogy « normálként » vagy « inaktiváltként » jelenítse meg őket az **m** és **p** változók állapotának függvényében, amik a *Tkinter* változók tükörképei.

Az **m** és **p** közbenső változók igazából csak a script érthetővé tételére szolgálnak. Kiküszöbölhetők és a scrip még tömörebbé tehető az utasítások kompozíciójával. A következő két utasítást :

```
m = self.actMusi.get()
self.musi.configure(state =[DISABLED, NORMAL][m])
```

például helyettesíthetnénk egy utasítással :

```
self.musi.configure(state =[DISABLED, NORMAL][self.actMusi.get()])
```

Jegyezzük meg, hogy amit a tömörséggel nyerünk, azért az olvashatóság bizonyos mértékű romlásával fizetünk.

d) Egy menüpont előválasztása

Gyakorlatunk befejezéseként nézzük még meg, hogyan lehet előre meghatározni bizonyos választásokat, illetve programból módósítani azokat.

Írjuk az **Application()** osztály constructorához a következő utasítást (például a **self.pack()** utasítás elé) :

```
mBar.mo.invoke(2)
```

Az így módosított script lefuttatásakor megállapítjuk, hogy induláskor a menüsor « Festők » menüpontja az aktív, míg a « Zenészek » az inaktív. Ahogyan programozva vannak, úgy alapértelmezetten mindkét menüpontnak aktívnak kell lenni. És valóban így van, ha töröljük a :

mBar.mo.invoke(2) utasítást.

Azért javasoltam ennek az utasításnak a scriptbe illesztését, hogy megmutassam, hogyan lehet ***programmal*** elérni ugyanazt, amit normálisan egy egérkattintással kapunk.

A fenti utasítás az **mBar.mo** widgetet ***hívja*** a widget második eleméhez kapcsolt parancsot működésbe hozva. A listát megnézve igazolhatjuk, hogy ez a második elem a *checkbutton* típusú objektum, ami aktiválja/inaktiválja a « Festők » menüt. (Ismét idézzük emlékezetünkbe, hogy a számozás mindíg nullával kezdődik.)

A program indításakor minden úgy történik, mintha a felhasználó először az « Opciók menü » « Festők » menüpontjára kattintana, aminek a hatása a megfelelő menü inaktiválása.

15. Fejezet : Konkrét programok elemzése

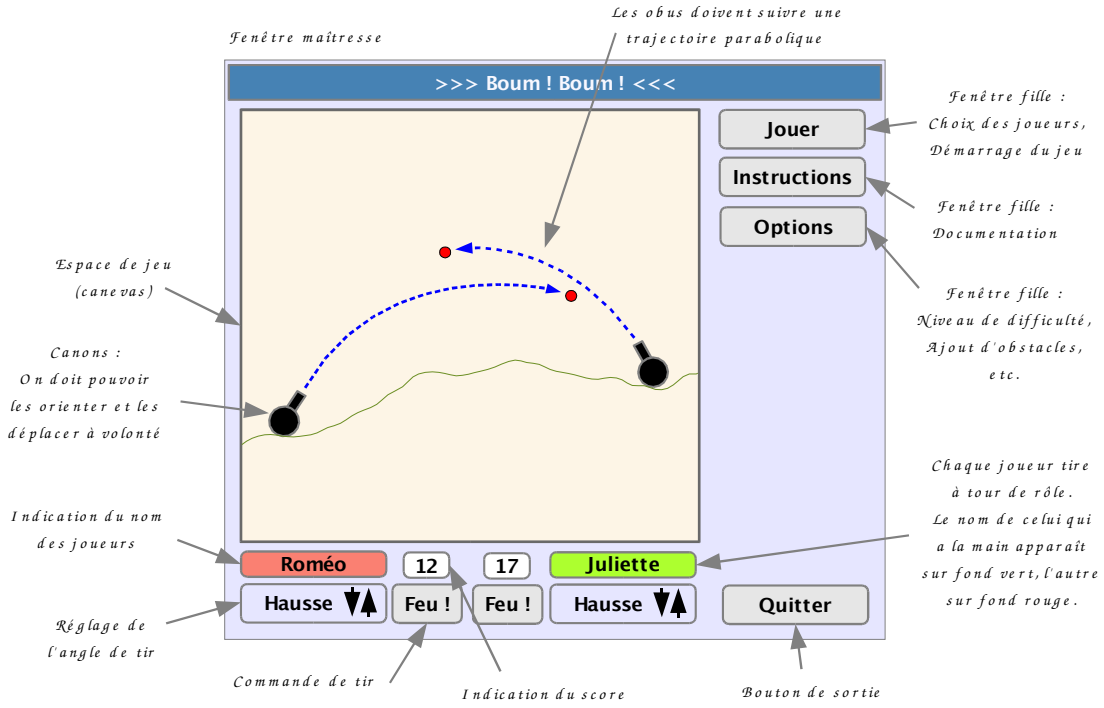
Ebben a fejezetben megpróbálom bemutatni egy grafikus program fejlesztésének a menetét az első vázlatoktól egészen egy viszonylag előrehaladott fejlesztési stádiumig. Így szeretném megmutatni, hogy az objektum orientált programozás mennyire le tudja egyszerűsíteni és főként milyen ***biztonságossá*** tudja tenni a javasolt inkrementális fejlesztési stratégiát[58].

Akkor kell osztályokat alkalmazni, amikor egy projekt a megvalósítása során lényegesen összetettebbnek mutatkozik, mint ahogy a kiinduláskor képzeltük. Biztosan az olvasó projektjei is az alábbiakban leírtakhoz hasonló fejlődési fokozatokon fognak átmenni.

15.1 Ágyúpárbaj

Ezt a projektet[59] a végzős tanulók egy hasonló munkája inspirálta.

Az ilyen projektek vázlatát rajzokkal és sémákkal érdemes kezdeni, amik a különböző létrehozandó grafikus elemeket és maximális számú ***use case***-t írnak le. Ha viszolygunk a papír és ceruza használatától, akkor használhatunk olyan rajzolóprogramot, mint amilyen az OpenOffice.org[60] Draw programja. Én ezt használtam az alábbi sémák elkészítéséhez :



A alapötlet egyszerű : két játékos áll egymással szemben egy-egy ágyúval. Mindegyiknek be kell állítani a lövés szögét és igyekezni kell eltalálni az ellenfelet. A lövedékek ballisztikus pályát írnak le.

---

58 Lásd : *Hibakeresés és kísérletezés* (16. oldal), valamint : *Ablakok menükkel* (218. oldal)

59 Azért beszéljük meg egy játékprogram fejlesztését, mert egy mindenki számára közvetlenül hozzáférhető területről van szó és a konkrét célok könnyen azonosíthatók. Magától értetődően ugyanezek a technikák felhasználhatók más « komolyabb » alkalmazások esetén is.

60 Ez az MS-Office-szal kompatibilis, komplett, szabad, ingyenes, irodai programcsomag. Linux, Windows, MacOS, Solaris ... alatt rendelkezésre áll. Ezt a könyvet teljes egészében a szövegszerkesztőjével készítettem. Letölthető a : *http://www.openoffice.org* website-ról.

Az ágyúk helye a játék elején véletlenszerűen van meghatározva (legalábbis magasságban). Az ágyúk minden lövés után elmozdulnak (azért, hogy fokozzuk a játék érdekességét, mert az ágyúk beállítása így nehezebb). A találatokat számoljuk.

Az előző oldal bevezető rajza képezheti az ***elemzés*** egyik formáját. Egy programprojekt fejlesztésének a megkezdése előtt mindíg törekednünk kell egy részletes ***feladat meghatározás*** készítésére. Ez az előtanulmány nagyon fontos. A kezdők többsége túlságosan gyorsan kezd el - egy homályos elképzelésből kiindulva - programot írni. Közben elhanyagolják a probléma struktúrájának a kutatását. Így azt kockáztatják, hogy programjuk kaotikussá válik, mert a struktúrát előbb, vagy utóbb helyére kell tenni. Gyakran azt tapasztalják, hogy törölniük kell és újra kell írniuk a projektjük ***túl monolitikus*** és/vagy ***rosszul paraméterezett*** részeit.

- ***Túl monolitikus*** : ez azt jelenti, hogy elmulasztottak egy komplex problémát több egyszerűbb részproblémára felbontani. Például függvények vagy osztályok hívása helyett több egymást követő összetett utasítást ágyaztak egymásba.
- ***Rosszul paraméterezett*** : ez azt jelenti, hogy csak egy speciális esetet kezeltek az általános eset helyett. Például egy grafikus objektumnak rögzítették a méreteit ahelyett, hogy az átméretezést lehetővé tevő változókról gondoskodtak volna.

Egy projekt fejlesztését mindíg a lehető legmélyebbre ható elemzéssel kell kezdeni és az elemzés eredményét egy olyan dokumentum együttesben (sémák, tervek, leírások) kell konkretizálni, ami a feladat meghatározást fogja alkotni. Nagyléptékű programok számára egyébként rendkívül kidolgozott analitikus módszerek léteznek (*UML, Merise...*). Ezeket nem írhatom itt le, mert egész könyvek tárgyát képezik.

Amikor ezt mondom, sajnos el kell fogadjam, hogy nagyon nehéz (sőt valószínűleg lehetetlen) megvalósítani egy programprojekt komplett analízisét a kiindulástól kezdve. Egy program csak akkor mutatja meg a gyengéit, amikor működni kezd. Ekkor állapítjuk meg, hogy maradtak olyan use case-ek és kényszerek, amiket a kiinduláskor nem láttunk előre. Másrészt egy programprojektet gyakorlatilag arra szánunk, hogy fejlődjön. Gyakran elő fog fordulni, hogy a fejlesztés során módosítani kell a feladat meghatározást és nem feltétlenül azért, mert a kiindulási elemzés hibás volt, hanem egyszerűen csak azért, mert kiegészítő funkcionalitásokat akarunk hozzátenni.

Mindíg a következő két szabály betartásával közelítsünk egy új programprojekthez :

- Az első kódsorok szerkesztése előtt mélységében írjuk le a projektünket. Minden erőnkkel törekedjünk arra, hogy nyilvánvalóvá tegyük mik a fő komponensei és milyen relációk kapcsolják őket össze (gondoljunk a programunk különböző use case-einek a leírására).
- Amikor hozzákezdünk programunk tényleges megvalósításához, kerüljük a túlságosan nagy utasításblokkok írását. Inkább bontsuk az alkalmazásunkat nagyobb számú, jól parametrizálható zárt komponensre úgy, hogy bármelyikük könnyen módosítható legyen a többi működésének veszélyeztetése nélkül és esetleg újra felhasználható legyen különböző összefüggésekben, ha a szükség úgy hozza.

Ennek az igénynek a kielégítésére találták ki az objektum orientált programozást .

Tekintsük például az előző oldalon lerajzolt vázlatot.

A kezdő programozó ennek a játéknak az elkészítését talán csak procedurális programozással fogja megkísérelni (vagyis elmulaszt új osztályokat definiálni). Egyébként magam is így jártam el végig a 8. fejezetben a grafikus interface-ek első megközelítése során. Ez az eljárás azonban csak a nagyon kis programok esetében igazolható (gyakorlatok és előzetes tesztek esetében). Amikor egy nagy projektbe kezdünk, a jelentkező problémák komplexitása gyorsan megnő és elkerülhetetlenné

válik a részproblémákra bontásuk.

Ennek a részproblémákra bontásnak a programeszköze az ***osztály*** (***classe***).

Talán jobban megértjük a hasznát, ha egy analógiát alkalmazok :

Minden elektomos berendezés kis számú alapkomponensből áll : tranzisztorokból, diódákból, ellenállásokból, kondenzátorokból, stb. Az első számítógépeket közvetlenül ezekből az alkatrészekből építették. Nagyméretűek, drágák voltak, ráadásul nagyon kevés funkciójuk volt és gyakran meghibásodtak.

Különböző technikákat fejlesztettek ki arra, hogy ugyanabba a dobozba bizonyos számú alapkomponenst integráljanak (zárjanak be). Ezeknek az új *integrált áramköröknek* a használatához már nem volt szükség a pontos felépítésük ismeretére : csak a globális funkciójuk volt érdekes. Az első integrált áramkörök még viszonylag egyszerűek voltak : logikai áramkörök, kapcsolók, stb. Ezeket egymással kombinálva olyan bonyolultabb jellemzőjű áramköröket kaptak, mint amilyenek a regiszterek és a dekóderek, amiket tovább lehetett integrálni és így tovább egészen a mai mikroprocesszorokig. Ezek több millió komponenst tartalmaznak, és a megbízhatóságuk mégis igen nagy marad.

Következésként, egy modern villamosmérnöknek, aki például egy bináris számlálót (ez egy olyan áramkör, ami bizonyos számú kapcsolót tartalmaz) akar készíteni nyilván jóval egyszerűbb, gyorsabb és biztosabb integrált áramkörökből válogatni, mint azzal fáradni, hogy többszáz tranzisztort és ellenállást építsen össze hiba nélkül.

Analóg módon, a modern programozó hasznát láthatja elődei összegyűlt munkáinak, amikor a Pythonban már rendelkezésre álló nagyszámú könyvtárba és osztályba integrált funkcionalitást használja. Még jobb, hogy a saját alkalmazása fő komponenseinek – különösen azoknak, amik több példányban jelennek meg az alkalmazásában - egységbezárására ő maga is könnyen létre tud hozni osztályokat. Az ilyen eljárás egyszerűbb, gyorsabb és biztosabb, mint a hasonló utasításblokkok megtöbbszörözése egy egyre nagyobb méretű és egyre kevésbé érthető monolitikus programban.

Vizsgáljuk meg például a vázlatunkat. A játék legfontosabb komponensei nyilván az ágyúk, amiket különböző helyeken és különböző irányokba kell megrajzolni. Legalább két pédány kell majd belőlük.

Ahelyett, hogy a játék során alkotórészeikből rajzolnánk fel a vászonra, az az érdekünk, hogy egységes programobjektumoknak tekintsük őket, amiknek számos tulajdonsága és reakciója (viselkedése) van. (Azt akarom ezzel kifejezni, hogy el kell legyenek látva különböző mechanizmusokkal, amiket speciális ***metódusok*** segítségével programból aktiválhatunk.) Biztos, hogy ésszerű rájuk szánni egy speciális osztályt.

15.1.1 A« Canon » (ágyú) osztály prototípusa

Egy ilyen osztály definiálásával több legyet ütünk egy csapásra. Nemcsak egy « dobozba » gyűjtjük össze a rajznak és az ágyú működésének megfelelő összes kódot - a program többi részétől külön, - hanem arra is lehetőséget adunk, hogy a játékban könnyen létrehozhassunk tetszőleges számú ágyút. Ez megnyitja a későbbi fejlesztések távlatát.

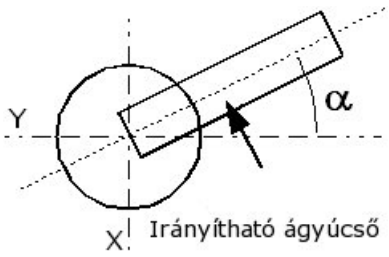
A **Canon()** (ágyú) osztály első implementációjának elkészítése és tesztelése során lehetőségünk lesz arra, hogy kiegészítő jellemzők hozzáadásával úgy tökéletesítsük az osztályt, hogy az interfaceét (az alkalmazási módját, azaz azokat az utasításokat, amik a különböző alkalmazásokban az osztály objektumainak létrehozásához és azok használatához szükségesek) ne, vagy csak kismértékben módosítsuk.

Vágjunk a dolgok közepébe.

Az ágyúnk rajzát rendkívül leegyszerűsíthetjük. Egy kör és egy téglalap kombinációjaként foghatjuk fel. A téglalapot egy vastag, egyenes szakasznak tekinthetjük.

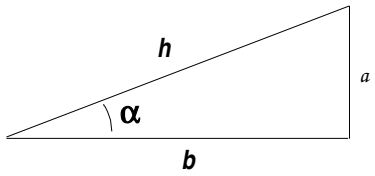
Ha egy színnel töltjük ki az ábrát (például feketével), akkor ágyúra emlékeztető alakot kapunk.

Meggondolásaink során azt feltételezzük, hogy az ágyú pozicióját a kör középpontja adja meg (az x és y koordináták a rajzon). Ez a pont adja meg az ágyúcső forgástengelyét is, valamint az ágyúcsövet reprezentáló szakasz egyik végét.



Rajzunk befejezéseként még meg kell határozni a szakasz másik végpontjának koordinátáit. Ha emlékszünk a trigonometriából a sinus és a cosinus fogalmára, akkor ez nem fog nehézséget okozni :

*Egy derékszögű háromszögben egy szöggel szemközti oldal és az átfogó hányadosa a szög **sinusa**. Ugyanennek a szögnek a **cosinus**-a szög melletti oldal és az átfogó hányadosa.*



Így a szemközti ábrában : $\sin\alpha = \frac{a}{h}$ és $\cos\alpha = \frac{b}{h}$.

Az ágyúcső reprezentálásához, feltéve, hogy ismerjük az ágyúcső **l** hosszát és a lövés α szögét, a kör (**x** és **y** koordinátájú) középpontjától kell húznunk egy vastag szakaszt, aminek a vízszintes hossza $\Delta x = l.\cos\alpha$ és a függőleges hossza $\Delta y = l.\sin\alpha$.

Összefoglalva a következőkből fog állni egy ágyú rajzolása az x, y pontba :

- rajzolunk egy x, y középpontú fekete kört
- egy vastag fekete szakaszt rajzolunk az x, y ponttól az x + l.cos α, y + l.sin α. pontig

Most elkezdhetünk gondolkodni a « Canon » osztálynak megfelelő programvázon. Itt még nem vetődik fel a játék programozásának kérdése. Csak azt akarjuk ellenőrizni, hogy az eddigi elemzésünk az első működőképes ***prototípus*** megvalósításának irányába mutat-e.

A prototípus egy elgondolás kikísérletezésére szánt programocska, amit azután egy nagyobb méretű alkalmazásba szándékozunk beépíteni. Az egyszerűsége és a tömörsége miatt a Python nagyon alkalmas prototípusok kidolgozására. Sok programozó használja különböző programkomponensek fejlesztésére, amiket aztán esetleg más « fajsúlyosabb » programnyelven, mint például a C, újraprogramoznak.

Első prototípusunkban a **Canon()** osztálynak csak két metódusa van : egy constructor, ami a rajz alapelemeit hozza létre és egy másik metódus, ami lehetővé teszi a lövés szögének (az ágyúcső dőlésszögének) beállítását. Ahogyan más példákban gyakran megtettem, most is teszek a script végéhez néhány kódsort, hogy rögtön tesztelni lehessen az osztályt :

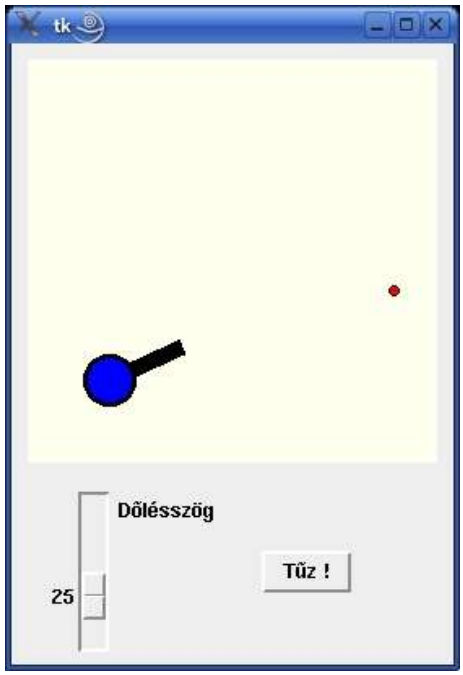
```
1.  from Tkinter import *
2.  from math import pi, sin, cos
3.
4.  class Canon:
5.      """Ágyúrajz"""
6.      def __init__(self, boss, x, y):
7.          self.boss = boss            # a vászon hivatkozása
8.          self.x1, self.y1 = x, y     # az ágyú forgástengelye
9.          # az ágyúcsövet vízszintesre rajzolja a kezdéshez :
10.         self.lagyucso = 50          # ágyúcső hossza
```

```
11.          self.x2, self.y2 = x + self.lagyucso , y
12.          self.cso = boss.create_line(self.x1, self.y1, self.x2, self.y2,
13.                                      width =10)
14.          # utána az ágyú testét felülre rajzolja :
15.          r = 15                        # a kör sugara
16.          boss.create_oval(x-r, y-r, x+r, y+r, fill='blue', width =3)
17.
18.      def iranyzas(self, angle):
19.          "az ágyúcső szögének kiválasztása"
20.          # rem : az <angle> paramétert stringként kapja.
21.          # valós számmá kell alakítani, utána radiánná :
22.          self.angle = float(angle)*2*pi/360
23.          self.x2 = self.x1 + self.lagyucso*cos(self.angle)
24.          self.y2 = self.y1 - self.lagyucso*sin(self.angle)
25.          self.boss.coords(self.cso, self.x1, self.y1, self.x2, self.y2)
26.
27.  if __name__ == '__main__':
28.      # kód a Canon osztály tesztelésére:
29.      f = Tk()
30.      can = Canvas(f,width =250, height =250, bg ='ivory')
31.      can.pack(padx =10, pady =10)
32.      c1 = Canon(can, 50, 200)
33.
34.      s1 =Scale(f, label='Dőlésszög', from_=90, to=0, command=c1.iranyzas)
35.      s1.pack(side=LEFT, pady =5, padx =20)
36.      s1.set(25)                            # kezdő szög
37.
38.      f.mainloop()
```

Magyarázatok :

- 6. sor : Abban a paraméterlistában, amit az objektum létrehozásakor a constructornak majd át kell adni, megadjuk az ágyú helyzetét a vásznon meghatározó x és y koordinátákat, a vászon referenciáját (a *boss* változó). Ez a referencia nélkülözhetetlen : a vászon metódusainak hívásakor fogjuk használni.
  Bevehetnénk még egy paramétert a lövés kezdőszögének kiválasztásához, de mivel egy specifikus metódust akarunk implementálni az irányzás szabályozására, ezért észszerűbb lesz azt a kívánt pillanatban hívni.
- 7. és 8. sor :Ezeket a változókat azokban a metódusokban fogjuk használni, amiket a későbbiekben fogunk kifejleszteni az osztályhoz. Példányváltozókká kell tennünk őket.
- 9. - 16. sorok : Először kirajzoljuk az ágyúcsövet, majd az ágyútestet. Így a cső egy része takarva marad. Ez lehetővé teszi az ágyútest esetleges színezését.
- 18. - 25. sorok : Ezt a metódust egy « angle » (szög) paraméterrel hívjuk, ami fokokban van megadva (és a vízszintestől mérjük). Ha a paraméter értékét egy *Entry* vagy *Scale* widget segítségével hozzuk létre, akkor azt string formájában fogjuk átadni a metódusnak, ezért mielőtt (az előző oldalon leírt) számolásra használnánk, valós számmá kell átalakítanunk.
- 27. - 38. sorok : Az új osztályunk teszteléséhez egy *Scale* widgetet használunk. A cursora kezdő pozíciójának definiálásához és így az ágyú kezdő dőlésszögének meghatározásához a **set()** metódusát kell hívni (36. sor).

15.1.2 Metódusok hozzáadása a prototípushoz



Prototípusunk működőképes, de nagyon kezdetleges. Tökéletesítenünk kell. Hozzá kell adnunk a lövedékek kilövésének képességét.

A lövedékeket « ágyúgolyókként » kezeljük : egyszerű kis körök lesznek, amik az ágyúcsővel megegyező irányban egy kezdősebességgel lépnek ki az ágyú torkából. Ahhoz, hogy valóságos röppályát írassunk le velük, most egy kicsit el kell mélyedjünk a fizikában :

Hogyan halad egy magára hagyott test, ha elhanyagolunk olyan másodlagos jelenségeket, mint amilyen a levegő közegellenállása ?

A probléma összetettnek tűnhet, de nagyon egyszerű a megoldása : elég ha elfogadjuk, hogy a lövedék egyidejűleg mozog vízszintesen és függőlegesen és ez a két mozgás egymástól független.

Egy animációs ciklust fogunk írni, amiben szabályos időközönként újra számoljuk a lövedék x és y koordinátáit, tudva hogy :

- A vízszintesen irányú mozgás ***egyenletes***. Minden iterrációban ugyanazt a Δx elmozdulást kell hozzáadni az x koordinátához.
- A függőleges irányú mozgás ***egyenletesen gyorsuló***. Ez azt jelenti, hogy minden iterrációban az y koordinátához egy növekvő Δy elmozdulást kell adnunk, ami mindíg ugyanazzal a mennyiséggel nő.

Nézzük meg ezt a scriptben :

A) Kezdésként a következő sorokat kell a constructor metódus végéhez írni. Ezek a « lövedék »-objektum és egy példányváltozó (ez az animáció megszakításához kell) létrehozására valók. Kiinduláskor minimális méretű lövedéket hozunk létre (egy 1 pixeles kört) azért, hogy szinte láthatatlan maradjon :

```
# lövedék rajzolása (animáció előtt egy pontra van redukálva) :
self.lovedek =boss.create_oval(x, y, x, y, fill='red')
self.anim =False                    # az animáció megszakítója
# a vászon szélességének és magasságának meghatározása :
self.xMax =int(boss.cget('width'))
self.yMax =int(boss.cget('height'))
```

A két utolsó sor a master *widget* (esetünkben ez a vászon) **cget()** metódusát használja a *widget* jellemzőinek meghatározására. Azt akarjuk, hogy a *Canon* osztály általános legyen, vagyis bármilyen környezetben újra felhasználható legyen. Ezért nem számolhatunk előre annak a vászonnak a méreteivel amin ezt az ágyút használni fogjuk.
Megjegyzés: A *Tkinter* ezeket az értékeket stringek formájában adja vissza. Ezért azokat numerikus típussá kell alakítani, ha a számolásainkban használni akarjuk őket.

B) Utána két új metódust kell hozzáadnunk : egyet a kilövésre és egy másikat arra, hogy a kilőtt golyót animáljuk :

```
1.      def tuz(self):
2.          "ágyúgolyó kilövése"
3.          if not self.anim:
4.              self.anim =True
5.              # a golyó indulóhelyzete (az ágyúcső vége) :
6.              self.boss.coords(self.lovedek, self.x2 -3, self.y2 -3,
7.                                              self.x2 +3, self.y2 +3)
8.              v =15              # kezdősebesség
9.              # ennek a sebbességnek a vízszintes és függőleges komponensei :
10.             self.vy = -v *sin(self.angle)
11.             self.vx = v *cos(self.angle)
12.             self.animal_lovedek()
13.
14.     def animal_lovedek(self):
15.         "golyó animációja (ballisztikus pálya)"
16.         if self.anim:
17.             self.boss.move(self.lovedek , int(self.vx), int(self.vy))
18.             c = self.boss.coords(self.lovedek )   # új koord. a mozgás után
19.             xo, yo = c[0] +3, c[1] +3   # lövedék középpontjának koord.
20.             if yo > self.yMax or xo > self.xMax:
21.                 self.anim =False        # animáció leállítása
22.             self.vy += .5
23.             self.boss.after(30, self.animal_lovedek)
```

Magyarázatok :

- 1. - 4. sorok : Ezt a metódust egy gombra történő kattintás hívja. Elindítja a lövedék mozgását és « igaz » értéket rendel az « animáció megszakításhoz » (a **self.anim** változóhoz : lásd a következőket). Biztosítanunk kell, hogy az animáció időtartama alatti újabb kattintás a gombra ne aktiválhasson más – parazita - animációs ciklusokat. A 3. sorban végrehajtott tesztnek ez a szerepe. Az utána következő utasításblokk csak akkor hajtódik végre, ha a **self.anim** változó értéke « hamis », ami azt jelenti, hogy az animáció még nem kezdődött el.
- 5. - 7. sorok : A *Tkinter* vászonnak két metódusa van a grafikai objektumok mozgatására :
  A **coords()** metódus abszolút pozícionálást végez; viszont meg kell neki adni az objektum összes adatát (mintha újra rajzolnánk). A 17. sorban használt **move()** metódus egy relatív elmozdítást végez; csak két argumentumot használ, a kívánt elmozdulás vízszintes és függőleges komponenseit.
- 8. - 12. sorok : A lövedék kezdősebességét a 8. sorban választjuk meg. Amint azt az előző oldalon magyaráztam, a lövedék mozgása egy vízszintes és egy függőleges mozgás eredője. Ismerjük a kezdősebességét és a szögét (a lövés szögét). A sebesség vízszintes és függőleges komponensének a meghatározásához hasonló trigonometrikus összefüggéseket kell használnunk, mint amilyeneket az ágyúcső rajzolásánál használtunk. A 9. sorban a – jel onnan ered, hogy a függőleges koordinátákat felülről lefelé számoljuk.
  A 12. sor aktiválja az animációt.
- 14 - 23. sorok : Ez az eljárás a 23. sorban hívott **after()** metódus révén 30 msec-onként újra hívja önmagát. Ez mindaddig ismétlődik, amíg a **self.anim** (« animáció megszakítás ») változó értéke « igaz ». Ez az érték meg fog változni amikor a lövedék koordinátái átlépik az előírt határokat (20. sor tesztje).
- 18.-19. sorok : Ahhoz, hogy minden egyes elmozdulás után meghatározzuk ezeket a koordinátákat, még egyszer hívjuk a vászon **coords()** metódusát : ha ennek egyetlen argumentuma van, ami egy grafikus objektum referenciája, akkor visszatérési értékként egy

tuple-ben megadja a grafikus objektum négy koordinátáját.

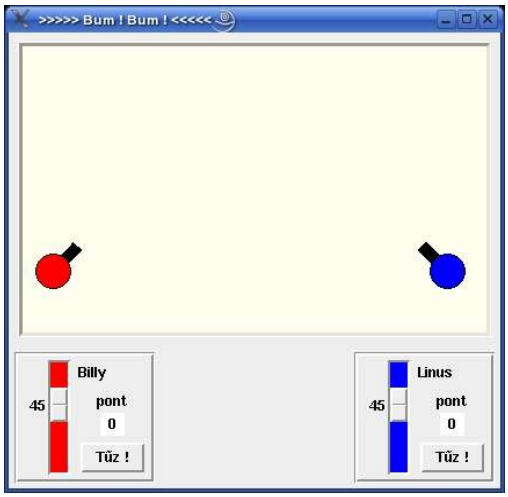
- 17. és 22. sorok : A lövedék vízszintes koordinátája mindíg ugyanannyival nő (egyenletes mozgás), míg a függőleges koordinátája egy olyan mennyiséggel növekszik, ami minden alkalommal maga is növekszik a 22. sorban (egyenletesen gyorsuló mozgás). Az eredmény egy parabolikus pálya.
  Emlékeztető : a **+=** operátor egy változót inkrementál : « a += 3 » azonos « a = a + 3 » -mal.

C) Végül egy indítógombot kell a főablakhoz adnunk. A következő sor (a tesztkódba kell beszúrni) tökéletesen megfelel erre a célra :

```
Button(f, text='Tűz !', command =c1.tuz).pack(side=LEFT)
```

15.1.3 Az alkalmazás fejlesztése

Mostmár, hogy van egy elég jól kidolgozott « canon » (ágyú) osztályunk, fontolóra vehetjük az alkalmazás kidolgozását. Mivel az objektum orientált programozás módszerének alkalmazása mellett döntöttünk, ezért ezt az alkalmazást olyan ***objekumok együtteseként*** kell elkészítenünk, ***amik a metódusaik közvetítésével hatnak egymásra***.



Természetesen számos objektum már létező osztályokból származik : mint a vászon, a gombok, stb. Az előző oldalakon láttuk, hogy az az érdekünk, hogy ezeknek az alapobjektumoknak jól meghatározott együtteseit minden olyan alkalommal új osztályokká egyesítsük, amikor ezeknek az együtteseknek a számára speciális funkcionalitást tudunk azonosítani. Ez volt a helyzet például a körök és a mozgó vonalak csoportja esetében, amikről úgy döntöttünk, hogy « canon » -nak (ágyúnak) fogjuk hívni.

Tudunk-e még azonosítani az induló projektünkben más komponenseket, amiket érdemes új osztályokba bezárni ? Biztosan. Például ott van a vezérlőpult, amit szeretnénk összekötni minden egyes ágyúval : ezen lehet összegyűjteni az ágyúcső dőlésszögét vezérlő eszközt, a tűzgombot, a találatszámlálót és esetleg még más kijelzéseket, mint amilyen például a játékos neve. Azért is érdemes neki külön osztályt kreálni, mert előre tudjuk, hogy két vezérlőpultra lesz szükség.

Természetesen ott van maga az alkalmazás. Ezt egy osztályba zárva elkészítjük a főobjektumunkat, ami az összes többi objektumot vezérelni fogja.

Most pedig elemezzük az alábbi scriptet. Megtaláljuk benne a továbbfejlesztett **Canon()** (ágyú) osztályt : hozzáadtam néhány attribútumot és három kiegészítő metódust, hogy kezelni lehessen az ágyúk elmozdulását és a találatokat.

Mostantól fogva az **Application()** osztály helyettesíti az előző prototípusok tesztkódját. Két **Canon()** és két **VezerloPult()** objektumot hozunk létre, amiket - a későbbi fejlesztést előre látva - szótárakba teszünk (elképzelhető az ágyúk és így a vezérlőpultok számának növelése). A játék most működőképes : az ágyúk minden lövés után elmozdulnak és a találatokat számlálják.

```
1.  from Tkinter import *
2.  from math import sin, cos, pi
3.  from random import randrange
4.
5.  class Canon:
6.      """Ágyúrajz"""
7.      def __init__(self, boss, id, x, y, irany, szin):
8.          self.boss = boss            # vászon hivatkozása
9.          self.appli = boss.master    # alkalmazásablak hivatkozása
10.         self.id = id                # ágyú azonosítója (chaîne)
11.         self.szin = szin            # ágyú színe
12.         self.x1, self.y1 = x, y     # ágyú forgástengelye
13.         self.irany = irany            # lövés iránya (-1:balra, +1:jobbra)
14.         self.lagyucso = 30               # ágyúcső hossza
15.         self.angle = 0              # alapértelm. dőlésszög(lövés szöge)
16.         # vászon magasságának és széleségének meghatározása :
17.         self.xMax = int(boss.cget('width'))
18.         self.yMax = int(boss.cget('height'))
19.         # ágyúcső rajzolása (vízszintes) :
20.         self.x2, self.y2 = x + self.lagyucso * irany, y
21.         self.cso = boss.create_line(self.x1, self.y1,
22.                                     self.x2, self.y2, width =10)
23.         # ágyútest rajzolásan (színes kör) :
24.         self.rc = 15                # kör sugara
25.         self.test = boss.create_oval(x -self.rc, y -self.rc, x +self.rc,
26.                                      y +self.rc, fill =szin)
27.         # rejtett lövedék előrajzolása (pont a vásznon kívül) :
28.         self.lovedek = boss.create_oval(-10, -10, -10, -10, fill='red')
29.         self.anim = False           # animáció indikátor
30.         self.explo = False          # robbanás indikátor
31.
32.     def iranyzas(self, angle):
33.         "ágyú dőlésszögének szabályozása"
34.         # megjegyzés: az <angle> paramétert stringként kapja meg.
35.         # valós értékké, majd radiánná kell alakítani :
36.         self.angle = float(angle)*pi/180
37.         # rem: inkább a coords metódust használjuk egészekkel :
38.         self.x2 = int(self.x1 + self.lagyucso * cos(self.angle) *
    self.irany)
39.         self.y2 = int(self.y1 - self.lagyucso * sin(self.angle))
40.         self.boss.coords(self.cso, self.x1, self.y1, self.x2, self.y2)
41.
42.     def elmozdit(self, x, y):
43.         "új x, y pozícióba viszi az ágyút"
44.         dx, dy = x -self.x1, y -self.y1     # elmozdulás értéke
45.         self.boss.move(self.cso, dx, dy)
46.         self.boss.move(self.test, dx, dy)
47.         self.x1 += dx
48.         self.y1 += dy
49.         self.x2 += dx
50.         self.y2 += dy
51.
52.     def tuz(self):
53.         "golyó kilövése – csak ha az előző befejezte a röptét"
54.         if not (self.anim or self.explo):
```

```
55.             self.anim =True
56.             # a jelenlévő összes ágyú leírásának összegyűjtése
57.             self.guns = self.appli.dictionnaireCanons()
58.             # a lövedék kiindulási pozíciója (ez az ágyú torka) :
59.             self.boss.coords(self.lovedek, self.x2 -3, self.y2 -3,
60.                                        self.x2 +3, self.y2 +3)
61.             v = 17              # kezdősebesség
62.             # kezdősebesség vízszintes és függőleges komponense :
63.             self.vy = -v *sin(self.angle)
64.             self.vx = v *cos(self.angle) *self.irany
65.             self.animal_lovedek()
66.             return True     # => jelzi, hogy elindult a lövedék
67.         else:
68.             return False    # => nem lehetett kilőni a lövedéket
69.
70.     def animal_lovedek(self):
71.         "lövedék animációja (ballisztikus pálya)"
72.         if self.anim:
73.             self.boss.move(self.lovedek, int(self.vx), int(self.vy))
74.             c = self.boss.coords(self.lovedek)     # úk koordináták
75.             xo, yo = c[0] +3, c[1] +3      # lövedék közepének koord.-ja
76.             self.test_akadaly(xo, yo)     # elértünk egy akadályt ?
77.             self.vy += .4                  # függőleges gyorsulás
78.             self.boss.after(20, self.animal_lovedek)
79.         else:
80.             # animáció vége – lövedék elrejtése és ágyúk elmozdulása :
81.             self.animacio_vege()
82.
83.     def test_akadaly(self, xo, yo):
84.         "teszteljük, hogy a lövedék elért- egy célt, vagy a játék határát"
85.         if yo >self.yMax or xo <0 or xo >self.xMax:
86.             self.anim =False
87.             return
88.         # megvizsgáljuk az ágyúk szótárát annak eldöntésére, hogy
89.         # valamelyik ágyú nincs-e közel a lövedékhez :
90.         for id in self.guns:              # id = kulcs a szótárban.
91.             gun = self.guns[id]           # megfelelő érték
92.             if xo < gun.x1 +self.rc and xo > gun.x1 -self.rc \
93.             and yo < gun.y1 +self.rc and yo > gun.y1 -self.rc :
94.                 self.anim =False
95.                 # lövedék robbanását rajzolja (sárga ) :
96.                 self.explo = self.boss.create_oval(xo -12, yo -12,
97.                                 xo +12, yo +12, fill ='yellow', width =0)
98.                 self.hit =id       # eltalált céltárgy referenciája
99.                 self.boss.after(150, self.robbanas_vege)
100.                break
101.
102.    def robbanas_vege(self):
103.        "robbanás törlése ; lövedék újrainicializálása ; pontszám kezelése"
104.        self.boss.delete(self.explo)    #  robbanás törlése
105.        self.explo =False               # új lövés engedélyezése
106.        # jelzi a sikert a master-ablaknak :
107.        self.appli.goal(self.id, self.hit)
108.
109.    def animacio_vege(self):
110.        "végrehajtandó akciók, amikor a lövedék elérte a röppályája végét"
111.        self.appli.agyuk_szetszorasa ()          # az ágyúk elmozdítása
112.        # lövedék elrejtése (a vásznon kívülre küldve) :
113.        self.boss.coords(self.lovedek, -10, -10, -10, -10)
114.
115.
116.class VezerloPult(Frame):
117.    """Egy ágyúhoz asszociált vezérlőpult"""
118.    def __init__(self, boss, canon):
119.        Frame.__init__(self, bd =3, relief =GROOVE)
120.        self.score =0
121.        self.appli =boss                # az alkalmazás hivatkozása
122.        self.canon =canon               # az asszociált ágyú hivatkozása
123.        # A lövés szögét vezérlő rendszer :
124.        self.beallitas =Scale(self, from_ =75, to =-15,
```

```
                        troughcolor= canon.szin, command =self.iranyzas)
        self.beallitas.set(45)               # a lövés kezdő szöge
        self.beallitas.pack(side =LEFT)
        # Az ágyút azonosító címke :
        Label(self, text =canon.id).pack(side =TOP, anchor =W, pady =5)
        # Tűzgomb :
        self.bTuz =Button(self, text ='Tűz !', command =self.tuzel)
        self.bTuz.pack(side =BOTTOM, padx =5, pady =5)
        Label(self, text ="pont").pack()
        self.pontok =Label(self, text=' 0 ', bg ='white')
        self.pontok.pack()
        # az ágyú irányának megfelelően balra vagy jobbra pozícionálni
        if canon.irany == -1:
            self.pack(padx =5, pady =5, side =RIGHT)
        else:
            self.pack(padx =5, pady =5, side =LEFT)

    def tuzel(self):
        "az asszociált ágyú tüzelésének indítása"
        self.canon.tuz()

    def iranyzas(self, angle):
        "az asszociált ágyú dőlésszögének beállítása"
        self.canon.iranyzas(angle)

    def pontHozzaadasa(self, p):
        "a pontszám növelése vagy csökkentése"
        self.score += p
        self.pontok.config(text = ' %s ' % self.score)

class Application(Frame):
    '''Az alkalmazás főablaka'''
    def __init__(self):
        Frame.__init__(self)
        self.master.title('>>>>> Bum ! Bum ! <<<<<')
        self.pack()
        self.jatek = Canvas(self, width =400, height =250, bg ='ivory',
                            bd =3, relief =SUNKEN)
        self.jatek.pack(padx =8, pady =8, side =TOP)

        self.guns ={}           # a jelen levő ágyúk szótára
        self.pult ={}           # a jelen levő vezérlőpultok szótára
        # 2 ágyú-objektum létrehozása (+1, -1 = ellentétes irányok) :
        self.guns["Billy"] = Canon(self.jatek, "Billy", 30, 200, 1, "red")
        self.guns["Linus"] = Canon(self.jatek, "Linus", 370,200,-1, "blue")
        # Ezekkel az ágyúkkal asszociált 2 vezérlőpult létrehozása
        self.pult["Billy"] = VezerloPult(self, self.guns["Billy"])
        self.pult["Linus"] = VezerloPult(self, self.guns["Linus"])

    def agyuk_szetszorasa(self):
        "az ágyúk véletlenszerű elmozdítása"
        for id in self.guns:
            gun =self.guns[id]
            # az ágyú irányától függően balra vagy jobbra mozdítás
            if gun.irany == -1 :
                x = randrange(320,380)
            else:
                x = randrange(20,80)
            # a tulajdonképpeni elmozdítás :
            gun.elmozdit(x, randrange(150,240))

    def goal(self, i, j):
        "az <i> ágyú jelzi, hogy eltalálta a <j> ellenfelet"
        if i != j:
            self.pult[i].pontHozzaadasa(1)
        else:
            self.pult[i].pontHozzaadasa(-1)

    def dictionnaireCanons(self):
        "a jelen levő"ágyúkat leíró szótárat adja meg
```

```
195.         return self.guns
196.
197. if __name__ =='__main__':
198.     Application().mainloop()
```

Magyarázatok :

- 7. sor : A prototípussal összehasonlítva, három paramétert adtunk a constructorhoz. Az **id** egy tetszőleges névvel azonosítja a **Canon()** osztály minden objektumát. A **irany** adja meg, hogy jobbra (irany = 1) vagy balra (irany = -1) lő az ágyú. A **szin** határozza meg az ágyú színét.
- 9. sor : Tudnunk kell, hogy minden *Tkinter widget*-nek van egy **master** attribútuma, ami az esetleges master *widget*-jük (konténerük) hívatkozását tartalmazza. Ez a hívatkozás tehát esetünkben a főalkalmazás hivatkozása. (Mi magunk egy hasonló technikát implementáltunk a **boss** attribútum segítségével a vászonra történő hívatkozáshoz).
- 42. - 50. sorok : Ez a metódus egy új pozícióba viszi az ágyút. Minden lövés után véletlenszerűen újra pozícionálja az ágyúkat, ami fokozza a játék érdekességét.
  Megjegyzés : a **+=** operátorral inkrementálhatjuk egy változó értékét :
  « a += 3 » és « a = a + 3 » jelentése ugyanaz.
- 56. és57. sorok : Megpróbáljuk úgy megkonstruálni a canon (ágyú) osztályunkat, hogy újra felhasználható legyen olyan nagyobb programokban, melyek tetszőleges számú ágyút tartalmazhatnak, amik a küzdelmek során megjelenhetnek és eltünhetnek. Ilyen távlatokból nézve, minden jelenlévő ágyúról minden egyes lövés előtt rendelkeznünk kell egy olyan leírással, amiből meg tudjuk határozni, hogy érte-e találat a célt. Ezt a leírást a főalkalmazás egy szótárban kezeli, aminek egy másolatát a **dictionnaireCanons()** metódusa hívásával kérhetjük.
- 66. - 68. sorok : Ugyanebben az általános perspektívában esetleg hasznos lehet a hívó programot arról informálni, hogy a lövedék valóban ki lett-e lőve vagy sem.
- 76. sor : A lövedék animációját mostantól fogva két kiegészítő metódus kezeli. A kód tisztázása érdekében egy külön metódusba (**test_akadaly()**) tettem azokat az utasításokat, amik annak meghatározására szolgálnak, hogy elértünk-e egy céltárgyat.
- 79. - 81. sorok : Az előzőekben láttuk, hogy a **self.anim** változóhoz « hamis » értéket rendelve megszakítjuk a lövedék animációját. Az **animal_lovedek()** metódus tehát abbahagyja önmaga hívását és végrehajtja a 81. sor kódját.
- 83. - 100. sorok : Ez a metódus határozza meg, hogy az aktuális lövedék koordinátái meghaladják-e az ablak szélének a koordinátáit vagy egy másik ágyú kordinátáihoz vannak közel. Mindkét esetben az animáció megszakítás aktiválódik, de a második esetben egy sárga színű « robbanást » rajzolunk és az eltalált ágyú hivatkozását tároljuk. Egy kis idő múlva a **robbanas_vege()** metódust hívja a feladat befejezéséhez, hogy a robbanási kört törölje és egy találatot jelző üzenetet küldjön a master ablaknak.
- 115 - 153. sorok : A **VezerloPult()** osztály a **Frame()** osztályból leszármaztatással definiál egy új *widget*-et egy olyan technikával, aminek most már ismerősnek kell lenni. Ez az új widget az ágyúmegdöntés és a tüzelés parancsait, valamint egy meghatározott ágyú találatszámának kiíratását tartalmazza. Az ágyú és a vezérlőpanel közötti vízuális megfeleltetést az azonos színük biztosítja. A vezérlőpanel **tuzeles()** és **iranyzas()** metódusai a kapcsolt **Canon()** objektummal az utóbbi metódusai révén kommunikálnak.
- 155 - 172. sorok : Az alkalmazásablak is egy **Frame()**-ből leszármaztatott *widget*. A constructora két ágyút és a vezérlőpaneljeiket hozza létre. Ezeket az objektumokat két szótárba teszi : a **self.guns** és a **self.pult** -ba. Ez aztán különböző szisztematikus műveletek végzését teszi

minegyikükön lehetővé (mint például a következő metódus). Ezzel az eljárással egyébként megőrizzük annak a lehetőségét, hogy a program későbbi fejlesztése során, ha szükséges, minden nehézség nélkül növelhetjük az ágyúk számát.

- 174. - 184. sorok : Minden lövés után ennek a metódusnak a hívására kerül sor. Véletlenszerűen mozdítja el az ágyúkat, ami növeli a játék nehézségét.

15.1.4 Kiegészítő fejlesztések

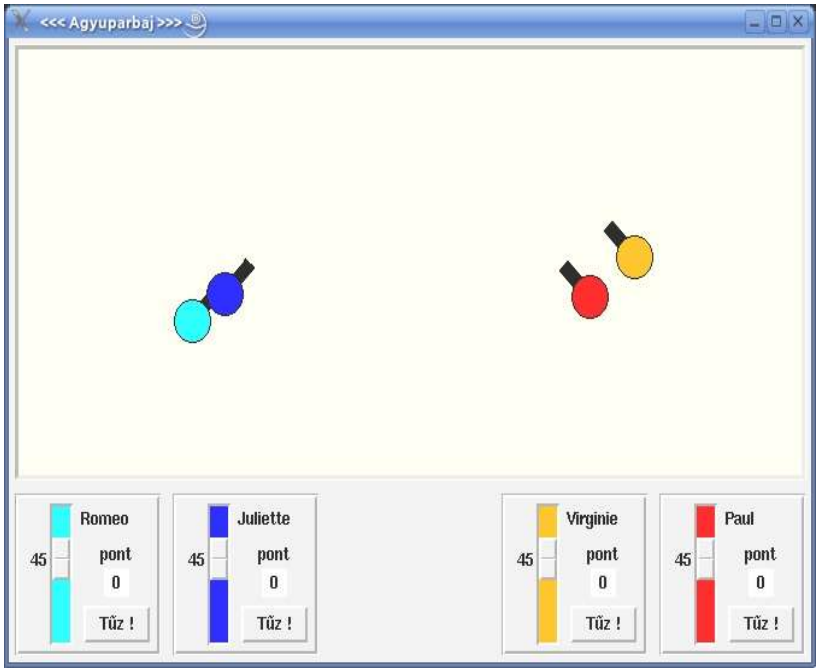
Programunk mostmár többé kevésbé megfelel a kiindulási feladat meghatározásnak, de nyilván folytathatjuk a tökéletesítését.

A) Jobban kellene például paramétereznünk. Ez mit jelent ? Mostani formájában a játékunk vászna előre meghatározott méretű (400 x 250 pixel, lásd 161. sort) Ha módosítani akarjuk ezeket az értékeket, akkor ügyelnünk kell rá, hogy a scriptnek azokat a sorait is módosítsuk, melyekben ezek a méretek szerepet játszanak (mint például a 168.-169., vagy a 179.-184. sorokat). Ha még más funkcionalitásokkal egészítjük ki a játékunkat, akkor azt kockáztatjuk, hogy megnő az ilyen egymástól függő sorok száma. Észszerűbb lenne változókkal méretezni a vásznat, amiknek az értékét egyetlen helyen definiálnánk. Ezeket a változókat aztán minden olyan utasítássorban használnánk, ahol a vászon mérete szerepet játszik.

Már elvégeztük ennek a munkának egy részét : a **Canon()** (ágyú) osztályban a vászon méreteit egy előre definiált metódussal nyertük vissza (lásd 17-18. sorokat) és olyan példány-attribútumokban helyeztük el, amiket az osztályban mindenütt használhatunk.

B) Minden lövés után az ágyúk random elmozdulását a koordinátáik véletlenszerű újradefiniálásával idézzük elő. Valószínűleg valósághűbb lenne valódi relatív ***elmozdulásokat*** kiváltani, mint az abszolút pozíciókat véletlenszerűen újradefiniálni. Ehhez át kell alakítani a **Canon ()** osztály **elmozdit()** metódusát. Még érdekesebb lenne úgy megcsinálni, hogy ez a metódus egy argumentumban átadott érték függvényében ugyanúgy tudjon relatív, mint abszolút elmozdulást előidézni.

C) A tűzparancsok rendszerét tökéletesíteni kellene : mivel csak egy egér van, meg kell kérni a játékosokat, hogy egymás után lőjenek, de semmilyen eljárást sem dolgoztunk ki, amivel erre kényszeríthetnénk őket. Az egy jobb megközelítés lenne, ha az ágyúdöntés és tűz parancsok bizonyos billentyűk lenyomásához lennének rendelve, amik a két játékos esetében különbözőek lennének.



D) Biztos, hogy programunk legérdekesebb továbbfejlesztése az lenne, ha ***hálózati alkalmazássá*** alakítanánk. A játék több egymással kommunikáló gépre lenne telepítve, mindegyik játékos egy ágyút irányítana. Egyébként még látványosabb lenne, ha megengednénk, hogy kettőnél több ágyúval játsszák úgy, hogy több játékos csatázhasson.

Az ilyen fajta fejlesztés viszont azt feltételezi, hogy már tökéletesen megtanultuk a programozás két olyan területét, amik meghaladják ennek a kurzusnak a kereteit :

- a ***socket*** technikát, ami két számítógép közötti kommunikáció létrehozását teszi lehetővé ;
- a ***thread*** technikát, ami lehetővé teszi, hogy ugyanaz a program több szimultán feladatot (task) hajtson végre (erre lesz szükségünk, ha olyan alkalmazást akarunk készíteni, ami egyidejűleg több partnerrel akar kommunikálni).

Szigorúan véve ezek a témakörök nem képezik ennek a kurzusnak a tárgyát, a tárgyalásuk önmagában egy egész fejezetet igényel. Ezért itt nem fogunk ezekbe belemenni. Azok megnyugtatására, akik érdeklődnek a tárgykör iránt : a könyv végén kiegészítésként megtalálható ez a fejezet (18. fejezet). Ott megtalálható az ágyúpárbaj hálózati verziója.

Addig viszont nézzük meg, hogyan fejleszthetjük tovább a projektünket úgy, hogy olyan javításokat hajtunk benne végre, amik 4 szereplős játékká alakítják. Arra is törekedni fogunk, hogy programunk olyan jól szegmentált legyen, hogy az osztályaink metódusai nagymértékben újra felhasználhatók legyenek. Menet közben meg fogjuk látni, hogyan valósítható meg ez a fejlesztés a már létező kód megváltoztatása nélkül úgy, hogy a már megírt osztályokból az ***öröklés*** alkalmazásával újakat hozunk létre

Kezdjük azzal, hogy az előző munkánkat elmentjük egy fileba, aminek a neve : **canon03.py**.

Így van egy Python modulunk, amit egyetlen utasítássorral importálhatunk egy új scriptbe. Ezt a technikát kihasználva folytatjuk alkalmazásunk tökéletesítését úgy, hogy csak az újdonságokat tartjuk szem előtt :

```
from Tkinter import *
from math import sin, cos, pi
from random import randrange
import canon03

class Canon(canon03.Canon):
    """Tökéletesített ágyú"""
    def __init__(self, boss, id, x, y, irany, szin):
        canon03.Canon.__init__(self, boss, id, x, y, irany, szin)

    def elmozdit(self, x, y, rel =False):
        "relatív elmozdulás, ha <rel> igaz; abszolút ha <rel> hamis"
        if rel:
            dx, dy = x, y
        else:
            dx, dy = x -self.x1, y -self.y1
        # vízszintes határok :
        if self.irany ==1:
            xa, xb = 20, int(self.xMax *.33)
        else:
            xa, xb = int(self.xMax *.66), self.xMax -20
        # csak ezek között a határok között mozdul el :
        if self.x1 +dx < xa:
            dx = xa -self.x1
        elif self.x1 +dx > xb:
            dx = xb -self.x1
        # függőleges határok :
        ya, yb = int(self.yMax *.4), self.yMax -20
        # csak ezek között a határok között mozdul el :
        if self.y1 +dy < ya:
            dy = ya -self.y1
        elif self.y1 +dy > yb:
            dy = yb -self.y1
        # az ágyúcső és a test elmozdulása :
        self.boss.move(self.cso, dx, dy)
        self.boss.move(self.test, dx, dy)
        # az új koordináták visszaadása a hívó programnak :
        self.x1 += dx
        self.y1 += dy
        self.x2 += dx
        self.y2 += dy
        return self.x1, self.y1

    def animacio_vege(self):
        "végrehajtandó akciók, amikor a lövedék elérte a röppályája végét"
        # annak az ágyúnak az elmozdítása, amelyik tüzelt :
        self.appli.agyu_veletlen_mozditasa(self.id)
        # lövedék elrejtése (a vásznon kívülre küldjük) :
        self.boss.coords(self.lovedek, -10, -10, -10, -10)

    def torol(self):
        "az ágyú eltüntetése a vászonról"
        self.boss.delete(self.cso)
        self.boss.delete(self.test)
        self.boss.delete(self.lovedek)

class AppAgyuParbaj(Frame):
    '''Az alkalmazás főablaka '''
    def __init__(self, larg_c, haut_c):
        Frame.__init__(self)
        self.pack()
        self.xm, self.ym = larg_c, haut_c
        self.jatek = Canvas(self, width =self.xm, height =self.ym,
                            bg ='ivory', bd =3, relief =SUNKEN)
        self.jatek.pack(padx =4, pady =4, side =TOP)

        self.guns ={}           # a jelen levõ ágyúk szótára
        self.pult ={}           # a jelen levõ vezérlõpultok szótára
        self.specificites()     # különb. objektumok a leszárm. oszt.okban

```

```
71.     def specificites(self):
72.         "ágyúk és vezérlőpanelek létrehozása"
73.         self.master.title('<<< Ágyúpárbaj >>>')
74.         id_list =[("Paul","red"),("Roméo","cyan"),
75.                   ("Virginie","orange"),("Juliette","blue")]
76.         s = False
77.         for id, szin in id_list:
78.             if s:
79.                 irany =1
80.             else:
81.                 irany =-1
82.             x, y = self.veletlen_koord(irany)
83.             self.guns[id] = Canon(self.jatek, id, x, y, irany, szin)
84.             self.pult[id] = canon03.VezerloPult(self, self.guns[id])
85.             s = not s           # az oldal megváltozt. minden iterrációban
86.
87.     def agyu_veletlen_mozditasa(self, id):
88.         "véletlenszerűen elmozdítja az <id> ágyút "
89.         gun =self.guns[id]
90.         dx, dy = randrange(-60, 61), randrange(-60, 61)
91.         # elmozdítás (az új koordináták meghatározásával) :
92.         x, y = gun.elmozdit(dx, dy, True)
93.         return x, y
94.
95.     def veletlen_koord(self, s):
96.         "véletlen koordináták, bal (s =1) vagy jobb (s =-1)"
97.         y =randrange(int(self.ym /2), self.ym -20)
98.         if s == -1:
99.             x =randrange(int(self.xm *.7), self.xm -20)
100.        else:
101.            x =randrange(20, int(self.xm *.3))
102.        return x, y
103.
104.    def goal(self, i, j):
105.        "la n°i ágyú jelzi, hogy eltalálta a n°j ellenfelet"
106.        # melyik táborhoz tartoznak ?
107.        ti, tj = self.guns[i].irany, self.guns[j].irany
108.        if ti != tj :               # ellenkező irányúak :
109.            p = 1                   # 1-pontot kapunk
110.        else:                       # azonos irányban vannak :
111.            p = -2                  # szövetségest találtunk el !!
112.        self.pult[i].pontHozzaadasa(p)
113.        # akit eltaláltak veszít egy pontot :
114.        self.pult[j].pontHozzaadasa(-1)
115.
116.    def dictionnaireCanons(self):
117.        "vissza adja a jelenlévő ágyúkat leíró szótárat"
118.        return self.guns
119.
120.if __name__ =='__main__':
121.    AppAgyuParbaj(650,300).mainloop()
```

Magyarázatok :

- 6. sor : A 4. sorban alkalmazott importálási forma lehetővé teszi egy új **Canon()** (ágyú) osztály definiálását, amit az előző **Canon()** osztályból származtatunk le, megőrizve az osztály nevét. Így azokat a kódrészeket, amik ezt az osztályt használják nem kell megváltoztatni. (Ez nem lett volna lehetséges, ha például a « from canon03 import * » formát haszáltuk volna.)
- 11. - 16 sor : Az itt definiált metódusnak ugyanaz a neve, mint a szülőosztály egyik metódusáé. Tehát az előbbi az új osztályban helyettesíteni fogja a szülőosztálytól örökölt azonos nevű metódust. (Azt fogjuk mondani, hogy az **elmozdit()** metódus ***overload***-olva van.) Amikor ilyen fajta módosításokat végzünk, akkor általában arra törekszünk, hogy ezeket úgy tegyünk, hogy az új metódus ugyanazt csinálja, mint a régi, amikor az új metódust a régivel megegyező módon hívjuk. Így biztosítjuk azt, hogy azok az alkalmazások, amik a szülőosztályt használják, a

gyermek osztályt is tudják majd használni anélkül, hogy az alkalmazást módosítani kellene.
Ezt az eredményt egy vagy több paraméter hozzáadásával érjük el, melyek alapértelmezett értékei a régi viselkedést idézik elő. Így amikor a **rel** paraméternek nem adunk meg semmilyen argumentumot sem, akkor az **x**-et és **y** -t abszolút koordinátákként használja (a metódus régi viselkedése). Viszont ha a **rel** -nek egy « igaz » argumentumot adunk meg, akkor az **x** és **y** paramétereket relatív elmozdulásokként kezeli (új viselkedés).

- 17. - 33. sorok : A kért elmozdulásokat véletlenszerűen állítjuk elő. Ezért gondoskodnunk kell egy programkorlátról, ami megakadályozza, hogy az objektum kimenjen a vászonról.
- 42. sorok : Az eredményül kapott koordinátákat visszaadjuk a hívó programnak. Lehet, hogy a hívó program az ágyú kezdő pozíciójának ismerete nélkül váltja ki az ágyú elmozdulását.
- 44. - 49. sorok : Itt megint egy szülőosztálytól örökölt metódus overload-járól van szó azért, hogy eltérő viselkedést kapjunk : lövés után mostmár nem mindegyik ágyút, hanem csak a lövést leadót mozdítjuk el.
- 51. - 55. sorok : Ezt a metódust olyan alkalmazások számára szánjuk, amik a játék során ágyúkat telepítenek, illetve távolítanak el.
- 57. és a következő sorok : Már a kezdetek kezdetén úgy tervezzük ezt az osztályt, hogy könnyen lehessen belőle más osztályokat leszármaztatni. Ezért szedjük szét a constructorát két részre : Az **__init__()** metódus tartalmazza azt a kódot, ami közös minden objektumra nézve, azokra is, amiket ebből az osztályból hozunk létre és azokra is, amiket egy esetleg leszármaztatott osztályból hozunk létre. A **specificites()** metódus tartalmazza a specifikusabb kódrészeket :ezt a metódust arra szánjuk, hogy overload-oljuk az esetleges leszármaztatott osztályokban.

15.2 A Ping játék

A következő oldalakon egy komplett program található. Példaként közlöm, amit az olvasó saját, összefoglaló projektjeként fejleszthet. A program ismételten bemutatja, hogyan konstruálhatunk több osztály felhasználásával egy jólstruktúrált scriptet.

15.2.1 Az elv

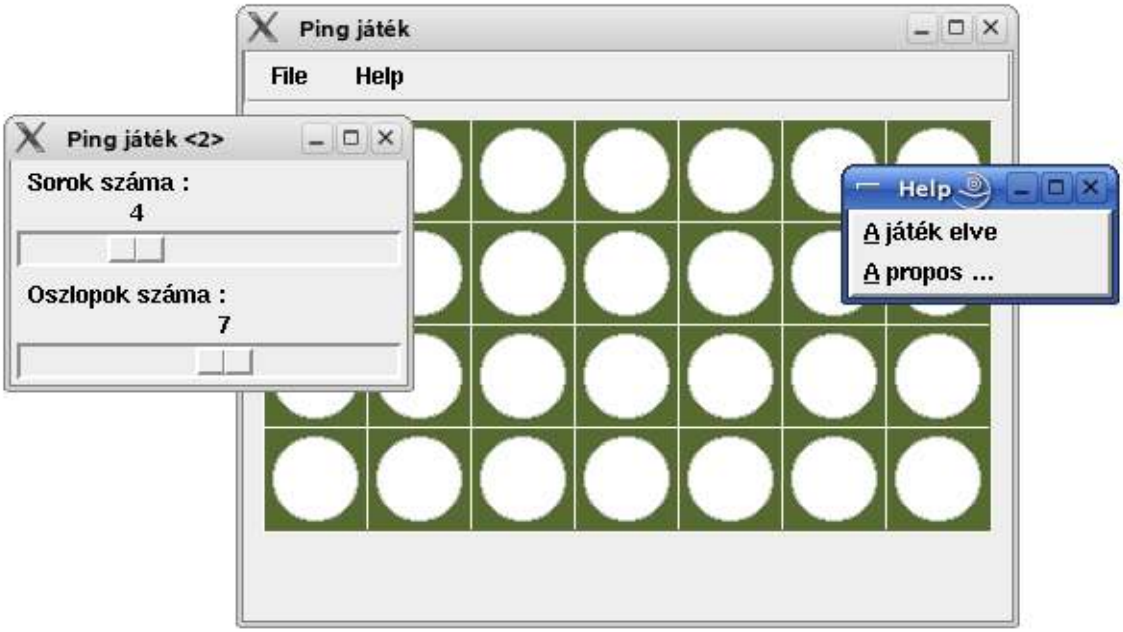
A « játék » inkább egyfajta matematikai gyakorlat. Táblán játszák, amin változó méretű négyzetrács jelenik meg. A rács mindegyik négyzetében egy korong van. A korongok mindegyikének van egy fehér és egy fekete oldala (mint az Othello/Reversi játékban). A játék kezdetekor mindegyiknek a fehér oldala van felül.

Amikor az egérrel rákattintunk az egyik korongra, a 8 szomszédos korong átfordul.

A játék abból áll, hogy bizonyos korongokra kattintva megpróbáljuk az összes korongot átfordítani.

A gyakorlat egy 2x2 -es négyzetráccsal nagyon egyszerű (elég mindegyik korongra rákattintani). Nagyobb rácsokkal nehezebbé válik, sőt bizonyos rácsokkal lehetetlen megoldani a feladatot. Az olvasó feladata annak meghatározása, hogy melyekkel !

(Ne hanyagoljuk el az 1 x n -es rács esetének vizsgálatát !



Megjegyzés : A Ping játék és elméletének tárgyalása megtalálható a: « Pour la science » n° 298 - Août 2002, p. 98 - 102 [*]

*A Scientific American francia szerkesztése. Szerzői jogi okból nem közölhetem a fordítását. A cikk az interneten hozzáférhető francia nyelven. (A fordító.)

15.2.2 Programozás

Egy programprojekt fejlesztésekor mindíg törekednünk kell a probléma megközelítésének a lehető legvilágosabb leírására. Kezdjük egy részletes feladat meghatározással és utána ne hanyagoljuk el a kódunk gondos kommentezését a kidolgozás során (nem pedig utána !)

Ha így járunk el, akkor arra kényszerítjük magunkat, hogy azt fogalmazzuk meg, amit szeretnénk, hogy a programunk csináljon. Ez segít a problémaelemzésben és a kód megfelelő struktúrálásában.

A fejlesztendő program feladatmeghatározása

- Az alkalmazást egy játéktáblát és egy menüsort tartalmazó főablakra alapozva fogjuk megkonstruálni.
- Az egésznek bővíthetőnek kell lenni a felhasználó szándéka szerint; a tábla rekeszeinek négyzet alakúnak kell maradniuk.
- A menüopciók tegyék lehetővé :
    - a rács méreteinek megválasztását (a négyzetek számának megadásával)
    - a játék újra inicializálását (vagyis minden korong fehér oldalával fölfelé van fordítva)
    - a játék elvének egy helpablakba történő kiírását
    - a befejezést (az alkalmazás bezárását)
- A program három osztályt fog hívni :
    - egy fő osztályt
    - egy menüsor osztályt
    - egy tábla osztályt
- A játéktábla egy vászonra lesz rajzolva, ami egy keretben (*frame*) lesz. A felhasználó által előidézett átméretezés függvényében a keret minden alkalommal ki fogja tölteni az egész rendelkezésre álló felületet : azaz úgy jelenik meg a programozónak, mint valamilyen téglalap, aminek a méretei kell, hogy alapul szolgáljanak a kirajzolandó rács méreteinek a számolásához.
- Mivel a rács rekeszeinek négyzet alakúnak kell maradni, ezért a maximális méretük számolásával az egyszerű kezdeni, majd ezek függvényében meghatározni a vászon méreteit.
- Az egérkattintás kezelése : a vászonhoz fogunk kapcsolni egy metódust, ami a <balgombra kattintás> eseményt kezeli. Az esemény koordinátái annak meghatározására fognak szolgálni, hogy a rács melyik rekeszébe (sorszám és oszlopszám) kattintottunk, bármilyen is legyen a rács mérete. A szomszédos 8 rekeszben lévő korongok át lesznek fordítva (a fekete és fehér színek cseréje).

```
##########################################
#  Ping játék                            #
#  Irodalom : Lásd a <Pour la science>   #
#  folyóirat, 2002 augusztusi számát     #
#                                        #
# (C) Gérard Swinnen (Verviers, Belgique) #
# http://www.ulg.ac.be/cifen/inforef/swi  #
#                                        #
#  Verzió 2002/09/29      - Licenc : GPL  #
##########################################

from Tkinter import *

class MenuBar(Frame):
    """Legordulo menusor"""
    def __init__(self, boss =None):
        Frame.__init__(self, borderwidth =2, relief =GROOVE)
        ##### <File> menu #####
        fileMenu = Menubutton(self, text ='Fichier')
        fileMenu.pack(side =LEFT, padx =5)
        me1 = Menu(fileMenu)
        me1.add_command(label ='Opciók', underline =0,
                        command = boss.options)
        me1.add_command(label ='Újrainditás', underline =0,
                        command = boss.reset)
        me1.add_command(label ='Vége', underline =0,
                        command = boss.quit)
        fileMenu.configure(menu = me1)

        ##### <Help> menu #####
        helpMenu = Menubutton(self, text ='Help')
        helpMenu.pack(side =LEFT, padx =5)
        me1 = Menu(helpMenu)
        me1.add_command(label ='A játék elve', underline =0,
                        command = boss.principe)
        me1.add_command(label ='A propos ...', underline =0,
                        command = boss.aPropos)
        helpMenu.configure(menu = me1)


class JatekTabla(Frame):
    """Jatektabla ( n x m es negyzetracs)"""
    def __init__(self, boss =None):
        # Ez a tabla egy atmeretezheto frame-bol all, amiben egy vaszon van
        # A frame minden atmeretezesekor kiszamoljuk a racs leheto legnagyobb
        # kockameretet es a vaszon meretet ennek megfeleloen allitjuk be

        Frame.__init__(self)
        self.nrow, self.ncol = 4, 4         # Kezdotabla = 4 x 4
        # A <resize> esemeny osszekapcsolasa egy megfelelo esemenykezelovel :
        self.bind("<Configure>", self.redim)
        # Canevas :
        self.can =Canvas(self, bg ="dark olive green", borderwidth =0,
                         highlightthickness =1, highlightbackground ="white")
        # Az <egerkattintas esemeny > osszekapcsolasa az esemenykezelojevel :
        self.can.bind("<Button-1>", self.clic)
        self.can.pack()
        self.initGame()

    def initGame(self):
        "A jatek allapotat tarolo lista inicializalasa"
```

```
        self.state =[]                  # egymasba agyazott lista letrehozasa
        for i in range(12):             # (egyenertekű egy 12 sor x 12 oszlopos
            self.state.append([0]*12)   #  tablaval)

    def redim(self, event):
        "atmeretezeskor vegrehajtando műveletek"
        # Az ujrakonfiguralas esemenyevel asszocialt tulajdonsagok
        # a frame uj mereteit tartalmazzak :
        self.width, self.height = event.width -4, event.height -4
        # A 4 pixeles különbseg a vasznat korülvevo 'highlightbordure'
        # kompenzalasara szolgal)
        self.drawGrid()

    def drawGrid(self):
        "A racs kirajzolasa az opciok es a meretek függvenyeben"
        # a negyzetek lehetseges max. szelessege es hosszusaga :
        lmax = self.width/self.ncol
        hmax = self.height/self.nrow
        # A negyzet merete ezen meretek legkisebbikevel egyenlo :
        self.side = min(lmax, hmax)
        # -> a vaszon uj mereteinek beallitasa :
        width_, height_ = self.side*self.ncol, self.side*self.nrow
        self.can.configure(width =width_, height =height_)
        # A racs kirajzolasa :
        self.can.delete(ALL)                # elozo rajzok torlese
        s =self.side
        for l in range(self.nrow -1):       # vizszintes vonalak
            self.can.create_line(0, s, width_, s, fill="white")
            s +=self.side
        s =self.side
        for c in range(self.ncol -1):       # függolees vonalak
            self.can.create_line(s, 0, s, height_, fill ="white")
            s +=self.side
        # Az osszes feher illetve fekete korong kirajzolasa a jatek allapotanak
        # megfeleloen :
        for l in range(self.nrow):
            for c in range(self.ncol):
                x1 = c *self.side +5            # korongok merete=
                x2 = (c +1)*self.side -5        # negyzetek merete -10
                y1 = l *self.side +5            #
                y2 = (l +1)*self.side -5
                colour =["white","black"][self.state[l][c]]
                self.can.create_oval(x1, y1, x2, y2, outline ="grey",
                                     width =1, fill =colour)

    def clic(self, event):
        "Az egerkattintas kezelese : a korongok megforditasa"
        # A sor es az oszlop meghatarozasaval kezdjük :
        row, col = event.y/self.side, event.x/self.side
        # Majd a szomszedos 8 negyzetet kezeljük :
        for l in range(row -1, row+2):
            if l <0 or l >= self.nrow:
                continue
            for c in range(col -1, col +2):
                if c <0 or c >= self.ncol:
                    continue
                if l ==row and c ==col:
                    continue
                # Korongforditas logikai inverzioval :
                self.state[l][c] = not (self.state[l][c])
        self.drawGrid()
```

```
class Ping(Frame):
    """A foprogram teste"""
    def __init__(self):
        Frame.__init__(self)
        self.master.geometry("400x300")
        self.master.title(" Ping jatek")

        self.mbar = MenuBar(self)
        self.mbar.pack(side =TOP, expand =NO, fill =X)

        self.game =JatekTabla(self)
        self.game.pack(expand =YES, fill=BOTH, padx =8, pady =8)

        self.pack()

    def options(self):
        "A racs sor es oszlopszamanak kivalasztasa"
        opt =Toplevel(self)
        curL =Scale(opt, length =200, label ="Sorok szama :",
              orient =HORIZONTAL,
              from_ =1, to =12, command =self.majLignes)
        curL.set(self.game.nrow)     # a cursor kezdo pozicioja
        curL.pack()
        curH =Scale(opt, length =200, label ="Oszlopok szama :",
              orient =HORIZONTAL,
              from_ =1, to =12, command =self.majColonnes)
        curH.set(self.game.ncol)
        curH.pack()

    def majColonnes(self, n):
        self.game.ncol = int(n)
        self.game.drawGrid()

    def majLignes(self, n):
        self.game.nrow = int(n)
        self.game.drawGrid()

    def reset(self):
        self.game.initGame()
        self.game.drawGrid()

    def principe(self):
        "A játék elvét tartalmazó üzenetablak"
        msg =Toplevel(self)
        Message(msg, bg ="navy", fg ="ivory", width =400,
            font ="Helvetica 10 bold",
            text ="A minden korongnak van egy fehér és egy fekete oldala "\
            "Amikor egy korongra kattintunk, a 8 szomszédja átfordul\n"\
            "A játék abból áll, hogy megpróbáljuk mindet átfordítani.\n\n"\
            "Ha a gyakorlat 2 x 2-es ráccsal nagyon egyszerűnek tűnik, "\
            "nehezebbé válik nagyobb méretű rácsokkal. Sőt bizonyos "\
            "méretű rácsokkal egyáltalán nem lehet lejátszani.!\n\n "\
            "Irodalom : 'Pour la Science' Aout 20022").pack(padx =10, pady =10)
```

```
    def aPropos(self):
        "A szerzőt és a licenc típusát megadó üzenetablak"
        msg =Toplevel(self)
        Message(msg, width =200, aspect =100, justify =CENTER,
            text ="Ping Játék\n\n(C) Gérard Swinnen, Aout 2002.\n"\
            "Licenc = GPL").pack(padx =10, pady =10)

if __name__ == '__main__':
    Ping().mainloop()
```

Emlékeztető : *Ha az olvasó az újraírásuk nélkül akar kísérletezni ezekkel a programokkal, akkor a forráskódjukat megtalálhatja a következő címen:*
http://www.ulg.ac.be/cifen/inforef/swi/python.htm

16. Fejezet : Adatbázis kezelés

Az adatbázisok egyre gyakrabban alkalmazott eszközök. Nagymennyiségű adat tárolását teszik lehetővé egyetlen jól struktúrált halmazban. Relációs adatbázisok esetén egyebek között lehetőség van a duplikált adatok elkerülésére is. Az olvasó biztosan találkozott már a következő problémával :

Azonos adatokat írtunk ki több file-ba. Ha módosítani, vagy törölni akarjuk egyiküket, akkor az összes filet, amelyik tartalmazza az adatot, meg kell nyitnunk és módosítanunk kell ! A hibázás veszélye nagyon reális, ami elkerülhetetlenül inkoherenciákhoz vezet, nem beszélve arról az időveszteségről, amit az adatmódosítás illetve törlés jelent.

Az adatbázisok jelentik **a** megoldást erre a problématípusra. A Python számos rendszer alkalmazását megengedi, de csak kettőt fogunk megvizsgálni : a ***Gadfly*** -t és a ***MySQL*** -t.

16.1 Adatbázisok

Számos adatbázistípus létezik. Például már elemi adatbázisnak tekinthetünk egy nevek és címek listáját tartalmazó filet.

Ha a lista nem túl hosszú és ha nem akarunk benne összetett feltételektől függő kereséseket végrehajtani, akkor magától értetődik, hogy ehhez az adattípushoz olyan egyszerű utasításokkal férhetünk hozzá, mint amilyeneket a 108. oldalon tárgyaltunk.

A helyzet azonban nagyon gyorsan komplikálttá válik, ha kiválasztásokat és rendezéseket hajtunk végre, különösen, ha ezek száma megnő. A nehézségek még tovább nőnek, ha az adatok különböző táblákban vannak, amiket hierarchikus relációk kapcsolnak össze és ha több felhasználónak egyidejűleg kell tudni hozzájuk férni.

Képzeljük például el, hogy az iskolánk vezetése megbíz bennünket egy számítógépes tanulmányi értesítő készítésével. Némi gondolkodás után rájövünk, hogy ez egy sor táblázat elkészítését feltételezi : egy tanulónévsor tábláét (ami természetesen más, a tanulókra vonatkozó specifikus információkat is tartalmaz : lakcím, születési dátum, stb.); egy tantárgytábláét (a tanár nevével, a heti óraszámmal, stb.); egy dolgozattábláét, amiben a dolgozatokat tároljuk az értékelésekhez (a dolgozat jellegével, dátumával, tartalmával, stb.); egy tanulócsoport tábláét, ami osztályonként és fakultációnként írja le a tanulócsoportokat, az egyes tanulók tantárgyait, stb., stb.

Az olvasó jól látja, hogy ezek a táblázatok nem függetlenek egymástól. Ugyanannak a tanulónak a dolgozatai különböző tantárgyakhoz kapcsolódnak. A tanuló tanulmányi értesítőjének elkészítéséhez természetesen adatokat kell kiszednünk a dolgozatok táblázatából, de ezeknek más táblákban talált információkkal is összefüggésben kell lenni (tantárgytábla, osztályok, opciók, stb.)

A későbbiekben meg fogjuk látni, hogyan írjuk le ezeket a táblákat és az őket összekötő kapcsolatokat.

16.1.1 Relációs adatbázis kezelő rendszerek – A kliens/server modell

Az ilyen komplex adatok hatékony kezelésére alkalmas programok maguk is szükségszerűen összetettek. Ezeket a programokat ***relációs adatbázis kezelő rendszereknek*** hívják (**RDBMS = *Relational Database Management Systems***). A vállalkozások számára elsődleges fontosságú alkalmazásokról van szó. Közülük egyesek speciálizálódott vállalatok (*IBM®, Oracle®, Microsoft®, Informix®, Sybase®*...) termékei és általában nagyon drágák. Másokat (*PostgreSQL©, MySQL©* ...) kutatási központokban vagy az egyetemi oktatásban fejlesztettek ki; ezek általában teljesen

ingyenesek.

E rendszerek mindegyike sajátos jellemzőkkel és teljesítménnyel rendelkezik, azonban a többségük működése a **kliens/server** modellen alapul. Ez azt jelenti, hogy az alkalmazás legnagyobb része (valamint az adatbázis) egyetlen helyre, elvileg egy nagyteljesítményű gépre van telepítve (ez az együttes alkotja a **server**), míg a másik jóval egyszerűbb rész meghatározatlan számú munkaállomásra van telepítve (ezeket hívjuk **kliens**eknek).

A kliensek különböző eljárások vagy protokolok (esetleg az internet) révén állandóan vagy ideiglenesen a serverhez vannak kapcsolva. Mindegyikük hozzáférhet az adatok több-kevesebb részéhez, egyes adatokat engedéllyel vagy anélkül módosíthatnak, újakat vihetnek be, törölhetnek a jólmeghatározott hozzáférési jogoktól függően. (Ezeket a jogosultságokat egy adatbázis **adminisztrátor** definiálja).

A server és kliensei különálló alkalmazások, amik információt cserélnek. Képzeljük például el, hogy egyike vagyunk egy rendszer felhasználóinak. Ahhoz, hogy hozzáférjünk az adatokhoz, valamelyik munkaállomáson el kell indítanunk egy kliens alkalmazást. A kliens alkalmazás azzal kezdi az indító processében, hogy kapcsolatot létesít a serverrel és az adatbázissal[61]. Amikor létrejött a kapcsolat, a kliens alkalmazás megfelelő formájú **kérés** (**request**) küldésével lekérdezheti a servert. Például egy meghatározott információt kell megkeresni. A server a megfelelő adatok adatbázisban történő keresésével végrehajtja a kérést, majd valamilyen **választ** küld vissza a kliensnek.

Ez a válasz lehet a kért információ, vagy hiba esetén egy hibaüzenet.

A kliens és a server közötti kommunikációt tehát kérések és válaszok alkotják. A kérések a klienstől a servernek küldött valódi utasítások, amik nemcsak adatok kinyerésére, hanem adatok beszúrására, törlésére, módosítására, stb. is szolgálnak.

16.1.2 Az SQL nyelv – Gadfly

Tekintettel arra, hogy különböző RDBMS-ek léteznek, azt hihetnénk, hogy mindegyikük esetében egy speciális nyelvet kell a nekik címzett kérésekhez használni. Mindenhol nagy erőfeszítéseket tettek egy közös nyelv kidolgozására és jelenleg létezik egy jól felépített standard : az **SQL** (***Structured Query Language***, vagy *struktúrált lekérdező nyelv*)[62].

Az olvasónak más tárgyak (pédául irodai alkalmazások) keretében valószínűleg alkalma lesz találkozni az SQL-lel. A Python-programozásba való bevezetés keretében két példa bemutatására fogok szorítkozni : kizárólag csak a Pythont felhasználva fogunk készíteni egy kis RDBMS-t és el fogjuk készíteni egy ambíciózusabb kliensprogram vázát, amit egy MySQL adatbázisserver-rel való kommunikációra szánok.

Első példánkban egy **Gadfly** nevű modult fogunk alkalmazni. Teljes egészében Pythonban van megírva. Ez a modul nem része a standard disztribúciónak, ezért külön kell telepíteni [63]. Az SQL parancsok egy nagy alcsoportját integrálja. Teljesítménye nyilván nem vethető össze a nagy,

61 A hozzáféréshez biztos, hogy meg kell adnunk néhény információt : a server hálózati címét, az adatbázis nevét, a felhasználó nevét, jelszót, ...

62 A különböző SQL implementációk között van néhány változat a nagyon specifikus lekérdezések számára, de az alap megegyezik.

63 A Gadfly modul ingyenesen hozzáférhető az interneten a következő címen : http://sourceforge.net/projects/gadfly
A telepítését a 17.6 függelék írja le a 309.oldalon.

specializált relációs adatbázis kezelő rendszerek teljesítményével[64], de szerény méretű adatbázisok kezelésére kiváló. Tökéletesen portábilis. A *Gadfly* ugyanúgy fog *Windows Linux* vagy *MacOS alatt* működni, mint a Python. Mitöbb, a *Gadfly* -val előállított adatbázisokat tartalmazó könyvtárakat ugyanúgy, módosítás nélkül használhatjuk egyik vagy másik operációs rendszerből.

Ha egy olyan alkalmazást akarunk fejleszteni, aminek viszonylag összetett relációkat kell kezelni egy kis adatbázisban, akkor a *Gadfly* modul nagyon leegyszerűsítheti a feladatot.

16.2 Egyszerű adatbázis készítése Gadfly-val

A következőkben megvizsgáljuk, hogyan készíthetünk egy egyszerű alkalmazást, ami ugyanazon a gépen egyszerre tölti be a server és a kliens szerepét.

16.2.1 Adatbázis létrehozása

Mint az olvasó bizonyára már várja, a megfelelő funkciókhoz való hozzáféréshez elég importálni a **gadfly** modult.
Utána létre kell hozni a gadfly egy példányát (egy objektumot) :

```
import gadfly
adatBazis= gadfly.gadfly()
```

Az így létrehozott **adatBazis** objektum a lokális adatbázismotorunk, ami a műveletek nagy részét a memóriában fogja végrehajtani. Ez a kérések nagyon gyors végrehajtását teszi lehetővé.

Az adatbázis létrehozásához az objektum **startup** metódusát kell hívni :

```
adatBazis.startup("mydata","E:/Python/essais/gadfly")
```

Az első paraméter, a **mydata**, az adatbázisnak választott név (természetesen más nevet is lehet választani !). A második paraméter az a könyvtár, ahová telepíteni akarjuk az adatbázist. (Ezt a könyvtárat előzőleg létre kell hoznunk. Minden a könyvtárban mydata néven már létező adatbázist figyelmeztetés nélkül meg fog semmisíteni a metódus.)

A beírt három kódsor elegendő : mostantól fogva van egy működőképes adatbázisunk, amiben létrehozhatunk különböző táblákat, majd ezekhez a táblákhoz adatokat adhatunk, adatokat törölhetünk, módosíthatunk.

Minden művelethez az SQL nyelvet fogjuk használni.

Ahhoz, hogy az **adatBazis** objektumnak át tudjuk adni az SQL kéréseinket, egy ***cursort*** kell készítenünk. Ez egyfajta memóriapuffer, ami az éppen kezelt adatok és a rajtuk végrehajtott műveletek átmeneti tárolására szolgál, mielőtt az adatokat véglegesen file-okba írnánk ki. Ez a technika lehetővé teszi, amennyiben szükséges, egy vagy több nem megfelelőnek bizonyult műveletet érvénytelenítését. (Az olvasó az SQL nyelvvel foglalkozó kézikönyvekből többet tudhat meg erről.)

Most vizsgáljuk meg az alábbi scriptet és jegyezzük meg, hogy az SQL kérések stringek, amiket a *cursor* objektum **execute** metódusa kezel :

```
cur = adatBazis.cursor()
cur.execute("create table membres (kor integer, nev varchar, meret float)")
cur.execute("insert into membres(kor, nev, magas) values (21,'Dupont',1.83)")
```

64 A Gadfly viszonylag hatékonynak látszik közepes méretű, egy felhasználós adatbázisok kezelésére. Többfelhasználós nagy adatbázisok kezelésére olyan RDBMS-eket kell alkalmazni, mint amilyen a PostgreSQL, Ezekhez is léteznek Python kliensmodulok (Pygresql például).

```
cur.execute("INSERT INTO MEMBRES(KOR, NEV, MAGAS) VALUES (15,'Suleau',1.57)")
cur.execute("Insert Into Membres(Kor, Nev, Magas) Values (18,'Forcas',1.69)")
adatBazis.commit()
```

Az első sor létrehozza a **cur** cursor-objektumot. A következő 4 sorban az idézőjelben lévő stringek az SQL kérések. ***Jól jegyezzük meg, hogy az SQL érzéketlen arra, hogy kis- vagy nagybetűt használunk*** : az SQL kéréseinket akár kis-, akár nagybetűkkel kódolhatjuk (természetesen ez nem áll fönn a környező Python-utasításokra !)

A második sor egy **membres** nevű adattáblát hoz létre, ami 3 mező bejegyzéseit fogja tartalmazni : az « egész szám » típusú **kor** mező, a « string » típusú (változó hosszúságú) **nev** mező és a « valós szám » típusú (lebegőpontos) **magas** mező bejegyzéseit. Az SQL nyelv elvileg más típusokat is megenged, de ezek nincsenek implementálva a ***Gadfly*** -ban.

A három következő sor hasonló. Összekevertem bennük a kis- és nagybetűket, hogy megmutassam, az SQL-ben ezek nem lényegesek. Ezek a sorok arra szolgálnak, hogy a **membres** táblába három bejegyzést szúrjunk be.

A műveleteknek ebben a fázisában a bejegyzések még nincsenek diszkfile-okba kiírva. Így vissza lehet térni a kiinduláshoz, amint azt a későbbiekben meg fogjuk látni. A diszkre történő adatkivitelt az utolsó utasítássor **commit()** metódusa aktiválja.

16.2.2 Kapcsolódás egy létező adatbázishoz

Tegyük fel, hogy a fenti műveletek során úgy határoztunk, befejezzük a scriptet vagy kikapcsoljuk a számítógépet. Hogyan kell eljárnunk a későbbiekben, hogy megint hozzáférjünk az adatbázisunkhoz ?

Ehhez két kódsorra van szükségünk :

```
import gadfly
adatBazis = gadfly.gadfly("mydata","E:/Python/essais/gadfly")
```

Ez a két sor elég a diszkre kiírt file-okban lévő táblázatok tartalmának a memóriába történő átviteléhez. Ettől kezdve az adatbázis lekérdezhető és módosítható :

```
cur = adatBazis.cursor()
cur.execute("select * from membres")
print cur.pp()
```

Az első sor megnyit egy cursort. A második sorban kiadott kérés egy bejegyzéscsoport kiválasztását kéri, ami az adatbázisból a cursorba fog átkerülni. Ebben az esetben a kiválasztás valójában nem egy bejegyzés kiválasztása : a membres tábla **összes** bejegyzését kérjük (az informatikában a * szimbólumot gyakran a « minden », « összes » jelentéssel használjuk).

A harmadik sorban a cursorra alkalmazott **pp()** metódus előformázott alakban kiíratja a cursor tartalmát (a megjelenített adatok automatikusan oszlopokba vannak rendezve). A « pp » jelentése « *pretty print* » (szép kiírás).

Ha magunk akarjuk irányítani az információk lapra tördelését, akkor a **pp()** metódus helyett a **fetchall()** metódust kell használnunk, ami egy tuple-kből álló listát ad vissza. Például próbáljuk ki a következőt :

```
for x in cur.fetchall():
    print x, x[0], x[1], x[2]
```

Természetesen további adatokat is bejegyezhetünk :

```
cur.execute("Insert Into Membres(Kor, Nev, Magas) Values (19,'Ricard',1.75)")
```

Egy vagy több bejegyzés módosításához az alábbi típusú kérést kell végrehajtani :

```
cur.execute("update membres set nev ='Gerart' where nev='Ricard'")
```

Egy vagy több bejegyzés törléséhez használjuk a következő kérést :

```
cur.execute("delete from membres where nev='Gerart'")
```

Ha mindezeket a műveleteket a Python parancssorában hajtatjuk végre, akkor az eredményt bármelyik pillanatban megnézhetjük egy « *pretty print* » -tel, ahogyan azt föntebb magyaráztam. Mivel a cursoron végrehajtott minden módosítás a memóriában történik, ezért semmi sem lesz végérvényesen rögzítve mindaddig, amíg végre nem hajtjuk az **adatBazis.commit()** utasítást.

Az előző **commit()** utasítás óta végrehajtott valamennyi módosítást törölhetjük úgy, hogy lezárjuk a kapcsolatot :

```
adatBazis.close()
```

16.2.3 Keresés egy adatbázisban

(16) Gyakorlat :

16.1. Aielőtt nagyon messzire mennénk, arra kérem az olvasót, hogy összefoglaló gyakorlatként hozzon létre teljesen önállóan egy « Zene » adatbázist, ami a következő két táblát tartalmazza (Ez némi munkába kerül, de adatokra van szükség ahhoz, hogy kísérletezni tudjon a kereső és rendező függvényekkel) :

muvek
szerzo (string)
cim (string)
ido (egesz)
eloado (string)

Zeneszerzok
szerzo (string)
ev_szul (egesz)
ev_halal (egesz)

Kezdje el feltölteni a **Zeneszerzok** táblát a következő adatokkal (használja ki az alkalmat és írjon egy kis scriptet, ami megkönnyíti az adatbevitelt : itt egy programhurkot érdemes alkalmazni !)

```
szerzo         ev_szul   ev_halal

Mozart         1756      1791
Beethoven 1770      1827
Handel         1685      1759
Schubert       1797      1828
Vivaldi        1678      1741
Monteverdi     1567      1643
Chopin         1810      1849
Bach           1685      1750
```

Írja be a **muvek** táblába a következő adatokat :

```
szerzo             cim                      ido       eloado

Vivaldi        Les quatre saisons           20        T. Pinnock
Mozart         Concerto piano N°12          25        M. Perahia
```

```
Brahms         Concerto violon N°2          40        A. Grumiaux
Beethoven Sonate "au clair de lune"    14        W. Kempf
Beethoven Sonate "pathétique"          17        W. Kempf
Schubert       Quintette "la truite"        39        SE of London
Haydn          La création                  109       H. Von Karajan
Chopin         Concerto piano N°1           42        M.J. Pires
Bach           Toccata & fugue              9         P. Burmester
Beethoven Concerto piano N°4           33        M. Pollini
Mozart         Symphonie N°40               29        F. Bruggen
Mozart         Concerto piano N°22          35        S. Richter
Beethoven Concerto piano N°3           37        S. Richter
```

A **ev_szul** és **ev_halal** mezők a zeneszerzők születésének és halálának az évszámát tartalmazzák. A zenemű időtartamát percekben adjuk meg. Nyilván annyi zeneművet és zeneszerzőt lehet rögzíteni, amennyit csak az olvasó akar, de az előbbi adatoknak elégnek kell lenni a kísérletezéshez.

A továbbiakban feltételezem, hogy a két fenti tábla adatai kódolva vannak. (Ha nehézségei vannak a script megírásával, úgy adok rá egy példát a függelékben a 356. oldalon).

A következő script csak tájékoztató jellegű. Egy alap SQL kliens, ami lehetővé teszi a « Zene » adatbázishoz – aminek már az egyik könyvtárunkban kell lenni – kapcsolódást, egy cursor megnyitását és a cursor alkalmazását a lekérdezések végrehajtására. Jegyezzük meg, hogy mindaddig amíg nem hívjuk a **commit()** metódust, addig semmi sem lesz kiírva a diszkre.

```
# SQL kéréseket elfogadó kis adatbázis használata

import gadfly

adatBazis = gadfly.gadfly("musique","E:/Python/essais/gadfly")
cur = adatBazis.cursor()
while 1:
    print "Írja be az SQL kérést (vagy <Enter> -t a befejezéshez) :"
    keres = raw_input()
    if keres =="":
        break
    try:
        cur.execute(keres)        # megkísérel végrehajtani SQL kérést
    except:
        print '*** INKORREKT KÉRÉS ***'
    else:
        print cur.pp()              # kérés eredményének kiírása
    print

valasztas = raw_input("Valóban mentsem (o/n) ? ")
if valasztas[0] == "o" or valasztas[0] == "O":
    adatBazis.commit()
else:
    adatBazis.close()
```

Ez a rendkívül egyszerű alakalmazás nyilván csak egy példa. Ki kellene egészíteni az adatbázis és a munkakönyvtár kiválasztásának lehetőségével. A 118. oldalon már leírt ***kivételkezelést*** alkalmaztam, hogy elkerüljem a script « kiakadását » amikor a felhasználó egy hibás kérést kódol.

16.2.4 A select utasítás

A **select** az SQL nyelv egyik leghatékonyabb utasítása. Ennek fogjuk most megnézni néhány alkalmazását. Ismétlem, a tárgykörnek csak nagyon kis részébe megyünk bele : az SQL részletes leírása több könyvet tenne ki.

Indítsuk el a fenti scriptet és elemezzük, hogy mi történik, amikor a következő kéréseket adjuk meg :

select * from muvek

select * from muvek where szerzo = 'Mozart'

select szerzo, cim, ido from muvek order by szerzo

select cim, szerzo from muvek where szerzo='Beethoven' or szerzo='Mozart' order by szerzo

select count(*) from muvek

select sum(ido) from muvek

select avg(ido) from muvek

select sum(ido) from muvek where comp='Beethoven'

select * from muvek where ido >35 order by ido desc

A kérések mindegyikénél igyekezzünk a lehető legjobban elmagyarázni, hogy mi történik. Lényegében szűrőket és rendezéseket aktiválunk az adatbázoson.

A következő kérések bonyolultabbak, mivel egyszerre két táblára vonatkoznak.

select o.cim, c.nev, c.ev_szul from muvek o, zeneszerzok c where o.szerzo = c.szerzo

select szerzo from muvek intersect select szerzo from zeneszerzok

select szerzo from muvek except select szerzo from zeneszerzok

select szerzo from zeneszerzok except select szerzo from muvek

select distinct szerzo from muvek union select szerzo from zeneszerzok

Nincs rá lehetőségem, hogy a jegyzet keretei között jobban kifejtsem a lekérdező nyelvet és általában az általános adatbázis kezelést. Viszont megvizsgálunk még egy Pythonnal készült példát, ami egy adatbázisrendszerhez fordul. Most azt tételezzük fel, hogy egy független server rendszerrel való párbeszédről van szó (ami például egy nagy, vállalati adatbázisserver, egy iskolai dokumentációs server, stb. lehet).

16.3 Egy MySQL kliensprogram váza

A fejezet befejezéseként a következő oldalakon egy konkrét megvalósításra mutatok be példát. Ez nem egy igazi program (a témakörnek egy könyvet kellene szentelni), inkább egy elemzésre való elnagyolt vázlat lesz, amivel az a célom, hogy bemutassam, hogyan gondolkodhatunk egy komplex probléma megközelítése során úgy, mint egy programozó.

A technikák, amiket majd kivitelezünk, egyszerű javaslatok. Megpróbáljuk bennük a legjobban felhasználni az előző fejezetekben megismert eszközöket, nevezetesen : magas szintű adatstruktúrákat (listákat és szótárakat), és az objektumokkal történő programozást. Magától értetődik, hogy a gyakorlatban tett választásaink nagymértékben kritizálhatók : ugyanezek a problémák nyilván más megközelítéssel is kezelhetők.

A konkrét célunk az, hogy gyorsan eljussunk egy olyan alapkliens kivitelezéséig, ami képes párbeszédet folytatni egy olyan « igazi » adatbázisserverrel, mint a MySQL. Azt szeretnénk, ha a kliensünk egy nagyon általános kis utility maradna, ami : képes egy több adattáblát tartalmazó kis adatbázis létrehozására, mindegyik táblába bejegyzéseket tud tenni, lehetővé teszi az alap SQL kérések eredményének tesztelését.

A következő sorokban feltételezem, hogy az olvasó már hozzáfér egy MySQL serverhez, amin a « jules » nevű felhasználónak, akinek a jelszava « abcde », létrehoztak egy « discotheque » adatbázist. Ez a server lehet egy hálózaton át elérhető távoli gépen, vagy lokálisan az olvasó PC-jén[65].

16.3.1 Az adatbázis leírása egy alkalmazáskönyvtárban

Egy adatbázissal párbeszédet folytató alkalmazás majdnem mindíg komplex alkalmazás. Nagyszámú kódsort tartalmaz. Ezeket osztályokba (vagy legalább függvényekbe) való zárással a lehető legjobban szeretnénk struktúrálni.

A kód számos helyén, amik gyakran nagyon távol vannak egymástól, az utasításblokkoknak figyelembe kell venni az adatbázis struktúrát, vagyis az adatbázis táblákra és mezőkre történő

---

65 Egy *MySQL* server installálása és konfigurálása meghaladja ennek a könyvnek a kereteit, de nem egy nagyon komplikált feladat. Sőt, nagyon egyszerű, ha olyan « kasszikus » disztribúciókból telepített Linux-szal dolgozunk, mint a *Debian*, *RedHat*, *SuSE* vagy *Mandrake.* Elegendő a *MySQL-server* és *Python-MySQL* package-eket telepíteni, elindítani a MySQL service-t, majd beírni a következő parancsokat :

```
mysqladmin  -u  root  password  xxxx
```

Ez a parancs a MySQL főadminisztrátorának jelszavát definiálja. A Linux rendszer adminisztrátorának ('root') kell végrehajtatni az általunk választott jelszóval. Majd az így definiált adminisztrátor accountjával kapcsolódunk a serverhez (a jelszót kérni fogja) :

```
mysql  -u  root  mysql  -p
grant  all  privileges  on  *.*  to jules@localhost  identified  by  'abcde';
grant  all  privileges  on  *.*  to jules@"%"  identified  by  'abcde';
\q
```

Ezek a parancsok egy új, « jules » nevű felhasználót definiálnak a MySQL rendszer számára és ehhez a felhasználóhoz kell kapcsolódni az « abcde » jelszónak. (A két sor a lokális és a hálózaton át történő hozzáférést engedélyezi.)

A felhasználónév tetszőleges : nem szükséges, hogy az operációs rendszer egy felhasználóneve legyen.

A « jules » nevű felhasználó rákapcsolódhat a serverre és adatbázisokat hozhat létre.

```
mysql  -u  jules  -p
create database discotheque;
\q
```

... stb. Ebben a stádiumban a MySQL server kész arra, hogy párbeszédet folytasson az ezeken az oldalakon leírt Python klienssel.

felosztását, valamint a bejegyzések között hierarchiát létesítő relációkat.

A gyakorlat azt mutatja, hogy egy adatbázis struktúrája ritkán végleges. A fejlesztés során gyakran jövünk rá, hogy be kell szúrni, vagy törölni kell mezőket, néha éppen egy rosszul felépített táblát kell helyettesíteni két másik táblával, stb. Nem bölcs dolog egy különleges struktúra túlságosan speciális kódrészeit « hard » kódolni.

Ezzel szemben ***inkább a programban egy helyen ajánlatos leírni az adatbázis komplett struktúráját*** és aztán ezt a leírást hívatkozásként használni az ilyen táblákra vagy mezőkre vonatkozó speciális utasítások félautomatikus generálásakor. Így javarészt elkerüljük annak a rémét, hogy minden alkalommal, amikor az adatbázis struktúrája bármilyen kis mértékben megváltozik, a kódban minden fele szétszórt nagyszámú utasítást kelljen behatárolni és módosítani. Ehelyett elegendő megváltoztatni a referencia leírást és a kód legnagyobb része korrekt marad, nem szükséges módosítani.

A stabil alkalmazások létrehozásának egyik vezérelvéhez ragaszkodunk :

Egy adatbázis kezelésre szánt programot mindíg egy alkalmazásszótár alapján kell megkonstruálni.

Amit itt « alkalmazásszótár » alatt értek, annak nem kell szükségszerűen egy Python szótár formáját felvenni. Bármilyen adatstruktúra megfelelhet, a lényeg, hogy egy adatokat leíró központi hívatkozás legyen, amit - esetleg a formázására vonatkozó információkkal – manipulálásra kínálunk fel.

Mivel a Python listák, tuple-k és szótárak bármilyen típusú adatokat képesek egy egységbe becsomagolni, ezért ezek tökéletesen megfelelnek erre a feladatra. A következő példában egy szótárat használunk, aminek az értékei tuple-k listái, de az olvasó a többi adatstruktúrát is ugyanígy választhatná ugyanezeknek az információknak az eltérő szervezésére.

Miután jól megszerveztük, még egy fontos kérdéssel kell foglalkoznunk : konkrétan hová fogjuk telepíteni az alkalmazásszótárat ?

Ezeket az információkat a program bármely részéről el kell tudnunk érni. Úgy tűnik, hogy egy globális változóba kell tenni az alkalmazásszótárat, pontosan úgy, mint más, a programunk egészének működéséhez szükséges adatokat. Márpedig tudjuk, hogy a globális változók használata nem javasolt : olyan veszélyeket hordoznak, amik a program méretével nőnek. Minden esetre a globálisnak nevezett változók csak ugyanazon a modulon belül globálisak. Ha a programunkat modulok együtteseként kívánjuk megszervezni (ami egyébként egy kitűnő gyakorlat), akkor globális változóinkhoz csak egyetlen modulban fogunk hozzáférni.

Erre a problémára van egy egyszerű és elegáns megoldás : ***külön osztályba kell összegyűjteni az alkalmazás együttese szempontjából globális státuszt igénylő változókat***. Így egy osztály névterébe bezárva ezek a változók probléma nélkül használhatók bármilyen modulban : elég, ha a modul importálja ezt az osztályt. Ráadásul ennek a technikának a használata egy érdekes következménnyel jár : az ilymódon definiált változók « globális » jellege világosan megjelenik a minősített nevükben, mert ez a név a változót tartalmazó osztály nevével kezdődik.

Ha például **Glob** a « globális » változóink számára létre hozott osztály neve, akkor a programunkban ezekre a változókra mindenütt a **Glob.ez** , **Glob.az**, stb [66].explicit nevekkel kell

66 « Globális » változóinkat elhelyezhetjük egy Glob.py nevű modulban, amit aztán importálunk. Modul vagy osztály névtérként való használata változók tárolására tehát hasonló technikák. Egy osztály használata talán egy kicsit rugalmasabb és olvashatóbb, mert az osztályt követheti a script többi része, míg egy modul szükségszerűen egy különálló file.

hivatkoznunk.

Ez a technika fedezhető fel scriptünk első soraiban. Definiálunk a scriptben egy **Glob()** osztályt, ami nem más, mint egy egyszerű konténer. Egyetlen objektumot sem fogunk létre hozni ebből az osztályból, ami nem tartalmaz metódusokat. A « globális változóink egyszerű osztályváltozókként vannak definiálva és így ezekre a program többi részében mint a **Glob** attribútumaira hivatkozhatunk. Például az adatbázis nevét mindenütt a **Glob.dbName** változóban lehet majd megtalálni, a server IP címét a **Glob.host** változóban, stb. :

```
1.  class Glob:
2.      """Névtér <pszeudo-global> változók és függvények számára"""
3.
4.      dbName = "discotheque"      # az adatbázis neve
5.      user = "jules"              # tulajdonos vagy felhasználó
6.      passwd = "abcde"            # hozzáférés jelszava
7.      host = "192.168.0.235"      # a server neve vagy IP címe
8.
9.      # Adatbázisszerkezet.  Táblák és mezők szótára :
10.     dicoT ={"Zeneszerzok":[('id_szerzo', "k", "elsődleges kulcs"),
11.                            ('nev', 25, "név"),
12.                            ('utonev', 25, "utónév"),
13.                            ('ev_szul', "i", "születési év"),
14.                            ('ev_halal', "i", "halál éve")],
15.             "muvek":[('id_mu', "k", "elsődleges kulcs"),
16.                      ('id_szerzo', "i", "szerző kulcs"),
17.                      ('cim', 50, "mű címe"),
18.                      ('ido', "i", "idő (percben)"),
19.                      ('eloado', 30, "előadó")]}
```

Az adatbázis szerkezetet leíró alkalmazásszótár a **Glob.dicoT** változóban van.

Ez egy Python szótár, aminek a kulcsai a táblanevek. Ami az értékeket illeti, mindegyikük egy lista, ami a tábla minden egyes mezőjének a leírását egy-egy tuple formájában tartalmazza.

Tehát mindegyik tuple a tábla egy külön mezőjét írja le. Hogy ne legyen a gyakorlat megterhelő, ezért összesen három információra korlátoztam a leírást : mezőnév, mezőtípus és rövid kommentár. Egy igazi alkalmazásban bölcs dolog lenne ezekhez hozzátenni még más információkat, amik például a mező adatainak esetleges határértékeire, a képernyőre vagy nyomtatóra történő kiírás formázására vonatkoznak; a szöveget, amit az oszlop fejlécébe kell írni, amikor az adatokat táblázatban akarjuk prezentálni, stb.

Az olvasónak meglehetősen unalmasnak tűnhet az adatstruktúra ennyire részletekbe menő leírása, amikor valószínűleg rögtön el szeretne kezdeni gondolkodni az adatok kezelésére szolgáló algoritmusok kivitelezésén. Egy jó struktúraleírással később biztosan sok időt fog nyerni, mert az számos dolog automatizálását teszi lehetővé. A későbbiekben ennek egy demonstrációját fogjuk látni. Egyébként meg kell róla győződnie, hogy ez a kicsit kellemetlen feladat felkészít arra, hogy a hátralévő munka jól struktúrált legyen : űrlapok szervezése, az elvégzendő tesztek, stb.

16.3.2 Egy interface-objektum osztály definiálása

Az előző szakaszban leírt **Glob()** osztályt a script elején helyezzük el, vagy egy modulban, amit a script elején importálunk. A továbbiakban feltételezem, hogy ezt az utóbbi formát alkalmazzuk : a **Glob()** osztályt egy **dict_app.py** nevű modulba mentettük, ahonnan most a következő scriptbe importálhatjuk.

Ez az új script egy interface-objektum osztályt definiál. Hasznot szeretnénk az előző fejezetekben tanultakból húzni, és ezért az objektumokkal való programozást támogatjuk, hogy

egységbe zárt és széles körben újra felhasználható kódrészleteket hozzunk létre.

Az interface-objektumok, amiket létre akarunk hozni, hasonlóak lesznek a 9. fejezetben filekezelésre használt fileobjektumokhoz. Emlékezzünk vissza : például úgy nyitunk meg egy filet, hogy az **open**() factory-függvény segítségével létrehozunk egy fileobjektumot. Hasonló módon fogjuk megnyitni a kommunikációt az adatbázissal. Az **ABKezelo**() osztály segítségével generálunk egy interface objektumot, ami majd létrehozza a kapcsolatot. A fileobjektum metódusait használjuk arra, hogy egy nyitott fileból olvassunk, illetve abba írjunk. Analóg módon egy interface-objektum különböző metódusaival fogunk műveleteket végrehejtatni az adatbázison.

```
1.  import MySQLdb, sys
2.  from dict_app import *
3.
4.  class ABKezelo:
5.      """Egy MySQL adatbázis létrehozása és illesztése"""
6.      def __init__(self, dbName, user, passwd, host, port =3306):
7.          "Kapcsolat létrehozása – Cursor létrehozása"
8.          try:
9.              self.adatBazis = MySQLdb.connect(db =dbName,
10.                   user =user, passwd =passwd, host =host, port =port)
11.         except Exception, err:
12.             print 'Nem sikerült a kapcsolat az adatbázissal :\n'\
13.                   'A detektált hiba :\n%s' % err
14.             self.kudarc =1
15.         else:
16.             self.cursor = self.adatBazis.cursor()   # cursor létrehozása
17.             self.kudarc =0
18.
19.     def createTables(self, dicTables):
20.         " <dicTables>.szótárban leírt táblák létrehozása"
21.         for table in dicTables:            # kulcsok bejárása.
22.             req = "CREATE TABLE %s (" % table
23.             pk =''
24.             for descr in dicTables[table]:
25.                 mezoNev = descr[0]        # létrehozandó mező címkéje
26.                 mt = descr[1]             # létrehozandó mező típusa
27.                 if mt =='i':
28.                     mezoTipus ='INTEGER'
29.                 elif mt =='k':
30.                     # 'elsődleges kulcs' mező (automatikusan incrementált)
31.                     mezoTipus ='INTEGER AUTO_INCREMENT'
32.                     pk = mezoNev
33.                 else:
34.                     mezoTipus ='VARCHAR(%s)' % mt
35.                 req = req + "%s %s, " % (mezoNev, mezoTipus)
36.             if pk == '':
37.                 req = req[:-2] + ")"
38.             else:
39.                 req = req + "CONSTRAINT %s_pk PRIMARY KEY(%s))" % (pk, pk)
40.             self.executeReq(req)
41.
42.     def deleteTables(self, dicTables):
43.         "<dicTables>-ben leírt összes tábla törlése"
44.         for table in dicTables.keys():
45.             req ="DROP TABLE %s" % table
46.             self.executeReq(req)
47.         self.commit()                       # kiíratás diszkre
48.
49.     def executeReq(self, req):
50.         "<req> kérés végrehajtása, az esetleges hiba detektálásával"
51.         try:
52.             self.cursor.execute(req)
53.         except Exception, err:
54.             # a kérés és a rendszer hibaüzenetének kiírása :
55.             print "Inkorrekt SQL kérés:\n%s\nDetektált hiba:\n%s"\
```

```
56.                     % (req, err)
57.             return 0
58.         else:
59.             return 1
60.
61.     def resultReq(self):
62.         "előző kérés eredményének visszaküldése (egymásba ágyazott tuplek)"
63.         return self.cursor.fetchall()
64.
65.     def commit(self):
66.         if self.adatBazis:
67.             self.adatBazis.commit()         # cursor -> diszk  átvitel
68.
69.     def close(self):
70.         if self.adatBazis:
71.             self.adatBazis.close()
```

Magyarázatok :

- 1.-2. sorok : A saját **dict_app** modulunk mellett, ami a « globális » változókat tartalmazza, importáljuk a : **sys** modult, ami néhány rendszerfüggvényt és a **MySQLdb** modult, ami mindazt tartalmazza, ami a **MySQL**-lel való kommunikációhoz szükséges. Emlékezzünk rá, hogy ez a modul nem képezi részét a standard Python disztribúciónak, ezért külön kell telepíteni.
- 5. sor : Az interface-objektumok létrehozásakor meg kell adnunk a kapcsolat paramétereit : az adatbázis nevét, a felhasználónevet, a felhasználó jelszavát, a gép nevét vagy az IP címét. A kommunikációs port száma szokás szerint az alapértelmezett érték. Feltételezem, hogy mindezek az információk rendelkezésre állnak.
- 8. - 17. sorok : Tanácsos a kommunikáció létrehozására szolgáló kódot egy ***try-except-else*** kivételkezelőben (lásd 119. oldalt) elhelyezni, mert nem tételezhetjük fel, hogy a server elérhető lesz. Jegyezzük meg, hogy az **__init__()** metódus nem adhat visszatérési értéket (a ***return*** utasítás segítségével) abból a tényből adódóan, hogy a Python automatikusan hívja egy objektum létrehozásakor. Amit ebben az esetben a hívó programnak visszaad, az az újonnan létrehozott objektum. Tehát a hívó programnak nem tudjuk jelezni egy visszatérési értékkel, hogy sikerült-e vagy pedig meghiúsult a kapcsolat létrehozása. Erre a problémára egy egyszerű megoldás, hogy a kapcsolatteremtés kísérletének az eredményét egy objektum-attribútumban (**self.kudarc** változó) tároljuk, amit a hívó program akkor tesztelhet, amikor az számára jónak tűnik.
- 19. - 40. sorok : Ez a metódus az alkalmazásszótár leírását – amit argumentumban kell neki megadni - kihasználva automatizálja az adatbázis tábláinak a létrehozását. Egy ilyen automatizálás nyilván annál jelentősebb, minél komplexebb lesz az adatbázis szerkezete. (Képzeljünk el például egy 35 adattáblát tartalmazó adatbázist !). Azért, hogy ne nehezítsem a bemutatást, az « ***integer*** » *és* « ***varchar*** » típusú mezők létrehozására korlátoztam ennek a metódusnak a képességeit. Az olvasó szabadon kiegészítheti más típusú mezők létrehozásához szükséges utasításokkal.

  Ha az olvasó alaposabban nézi meg a kódot, meg fogja állapítani, hogy az minden tábla számára egy - a **req** stringbe darabról-darabra történő - SQL kérés összeállításából áll. Az SQL kérést aztán argumentumként végrehajtásra átadjuk az **executeReq()** (kérés végrehajtása) metódusnak. Ha meg akarjuk nézni az így létrehozott kérést, nyilván a 40. sor után a kódhoz lehet írni egy « **print req** » utasítást.

  Ehhez a metódushoz hozzá lehet adni ***a hivatkozási integritási kényszerek*** bevezetésének a képességét is egy alkalmazásszótárban lévő kiegészítés alapján, ami leírja ezeket a kényszereket.

Nem fejtem ki a kérdést, de ha az olvasó tudja hogy miről van szó nem kellene ennek problémát okozni.

- 42. - 47. sorok : Ez a metódus sokkal egyszerűbb, mint az előző, ugyanazt az elvet alkalmazza az alkalmazásszótárban leírt összes tábla törlésére.
- 49. - 59. sorok : A metódus átadja a kérést a cursorobjektumnak. A haszna az, hogy egyszerűsíti a hozzáférést a cursor-objektumhoz és ha szükséges hibaüzenetet generál.
- 61. - 71. sorok : Ezek a metódusok csak egyszerű közvetítők a MySQLdb modul által létrehozott objektumok felé : a **MySQLdb.connect**() factory-függvény által létrehozott connector-objektum és a megfelelő cursor-objektum felé. Lehetővé teszik a hívó program némi egyszerűsítését.

16.3.3 Form-generátor készítése

Azért adtam ezt az osztályt a gyakorlathoz, hogy bemutassam, hogyan használhatjuk fel ugyanazt az alkalmazásszótárat egy általános kód kidolgozására. Az az elképzelés, hogy egy form-objektum osztályt konstruálunk, ami bármilyen adattábla bejegyzés kódolásáról képes gondoskodni az alkalmazásszótárból kinyert információk alapján automatikusan létrehozott megfelelő adatbeviteli utasítások révén.

Egy valódi alkalmazásban biztosan jelentősen át kellene alakítani ezt a végletekig leegyszerúsített formot. Valószínűleg specializált ablak alakja lenne, amiben az adatbeviteli mezőket és címkéiket megint csak automatikusan hozhatnánk létre. Nem állítom, hogy ez jó példa, pusztán azt akarom vele megmutatni, hogyan automatizálható a form konstrukciója. Hasonló elveket használva próbálja meg az olvasó elkészíteni a saját formjait.

```
1.  class adatRogzites:
2.      """különböző bejegyzések kezelésére szolgáló osztály"""
3.      def __init__(self, bd, table):
4.          self.bd =bd
5.          self.table =table
6.          self.description =Glob.dicoT[table]   # mezők leírása
7.
8.      def enter(self):
9.          "bevitt adat rögzítése az adatbázisban"
10.         mezok ="("           # vázstring a mezőneveknek
11.         ertekek ="("         # vázstring az értékeknek
12.         # Egymás után minden mezőnek értéket kérünk :
13.         for cha, type, nom in self.description:
14.             if type =="k":    # nem fogjuk kérni a bejegyzés sorszámát
15.                 continue      # a felhasználótól (automatikus sorszámozás)
16.             mezok = mezok + cha + ","
17.             val = raw_input("Írja be az értéket %s :" % nev)
18.             if type =="i":
19.                 ertekek = ertekek + val +","
20.             else:
21.                 ertekek = ertekek + "'%s'," % (val)
22.
23.         mezok = mezok[:-1] + ")"      # az utolsó vessző törlése
24.         ertekek = ertekek[:-1] + ")"  # egy zárójel hozzáadása
25.         req ="INSERT INTO %s %s VALUES %s" % (self.table, mezok, ertekek)
26.         self.bd.executeReq(req)
27.
28.         ch =raw_input("Folytatja (I/N) ? ")
29.         if ch.upper() == "I":
30.             return 0
31.         else:
32.             return 1
```

Magyarázatok :

- 1. - 6. sorok : Ennek az osztálynak az objektumai létrehozásuk pillanatában megkapják a szótár egyik táblájának a hivatkozását. Ennek segítségével férnek hozzá a mezőleírásokhoz.
- 8. sor : Az **enter()** metódus generálja a formot. Ő gondoskodik a bejegyzéseknek az adattáblába való beírásáról a szótárban talált leírás segítségével alkalmazkodva azok struktúrájához.
  A működése megint abból áll, hogy részekből megalkot egy stringet, ami egy SQL kérés lesz, ahogyan az az előző szakaszban az **ABKezelo()** osztály **createTables()** metódusában le van írva.

  Természetesen még más - például a bejegyzések törlésére és/vagy módosítására szolgáló - metódusokat is hozzáadhatnánk ehhez az osztályhoz.
- 12. - 21. sorok : A **self.description** példány-attribútum egy tuple-kből álló listát tartalmaz. A tuple-k mindegyike három elemből áll, ezek : a mező neve, a várt adat típusa és « világos » leírása. A 13. sor **for** ciklusa bejárja ezt a listát és mindegyik mezőhöz kiír a mezőt kísérő leírás alapján egy adatbevitelre felszólító üzenetet. Amikor a felhasználó beírta a kért értéket, akkor az egy stringben meg lesz formázva. A formázás a mező típusának megfelelően alkalmazkodik az SQL nyelv konvencióihoz.
- 23. - 26. sorok : Amikor minden mezőt bejártunk, a kérést összeállítja és végrehajtja. Ha meg akarjuk nézni a kérést, természetesen a 25. sor után írhatunk egy « **print req** » utasítást.

16.3.4 Az alkalmazás teste

Egy kezdők számára írt könyvben nem látom értelmét a gyakorlat továbbfejlesztésének. Ha a tárgykör érdekli az olvasót, akkor már eleget kell tudnia ahhoz, hogy maga is el tudjon kezdeni kísérletezni. Meg kell nézni olyan jó referenciaműveket, mint Deitel és tsai « *Python : How to program* »-ja , vagy a Python bővítményekről szóló websiteokat.

A következő script egy kis alkalmazás, ami az előző oldalakon leírt osztályok tesztelésére való. Tökéletesíthető vagy egy másik, teljesen eltérő script írható belőle !

```
1.  ###### Főprogram : #########
2.
3.  #  Adatbázis-interface objektum létrehozása :
4.  bd = ABKezelo(Glob.dbName, Glob.user, Glob.passwd, Glob.host)
5.  if bd.kudarc:
6.      sys.exit()
7.
8.  while 1:
9.      print "\nMit akar csinálni :\n"\
10.           "1) Adattáblákat létrehozni\n"\
11.           "2) Adattáblákat törölni ?\n"\
12.           "3) Zeneszerzőket beírni\n"\
13.           "4) Műveket beírni\n"\
14.           "5) Zeneszerzőket listázni\n"\
15.           "6) Műveket listázni\n"\
16.           "7) Valamilyen SQL kérést végrehajtani\n"\
17.           "9) Kilépni ?                          Válasszon :",
18.     ch = int(raw_input())
19.     if ch ==1:
20.         # a szótárban leírt összes tábla létrehozása :
21.         bd.createTables(Glob.dicoT)
22.     elif ch ==2:
23.         # a szótárban leírt összes tábla törlése :
24.         bd.deleteTables(Glob.dicoT)
25.     elif ch ==3 or ch ==4:
26.         # zeneszerzők vagy művek adatRogzites objektumának létrehozás :
27.         table ={3:'zeneszerzok', 4:'muvek'}[ch]
28.         enreg =adatRogzites(bd, table)
29.         while 1:
```

```
30.             if enreg.enter():
31.                 break
32.     elif ch ==5 or ch ==6:
33.         # az összes zeneszerző vagy az összes mű listázása :
34.         table ={5:'zeneszerzok', 6:''}[ch]
35.         if bd.executeReq("SELECT * FROM %s" % table):
36.             # a fenti kérés eredményének elemzése :
37.             records = bd.resultReq()      # tuple-kből álló tuple lesz
38.             for rec in records:             # => minden bejegyzés
39.                 for item in rec:            # => minden mező a bejegyzésben
40.                     print item,
41.                 print
42.     elif ch ==7:
43.         req =raw_input("Írja be az SQL kérést : ")
44.         if bd.executeReq(req):
45.             print bd.resultReq()          # tuple-kből álló tuple lesz
46.     else:
47.         bd.commit()
48.         bd.close()
49.         break
```

Magyarázatok :

- Természetesen fel fogom tételezni, hogy a föntebb leírt osztályok is benne vannak ebben a scriptben vagy importáltuk őket.
- 3. - 6. sorok : Az interface-objektumot hozzuk itt létre. Ha ez nem sikerül, akkor a **bd.kudarc** objektum-attribútum az 1 értéket tartalmazza. Az 5. és 6. sorok tesztje az alkalmazás azonnali leállítását teszi lehetővé (a **sys** modul **exit()** függvénye speciálisan erre szolgál).
- 8. sor : Az alkalmazás hátralévő része egy végtelen hurok, ami mindaddig ugyanazt a menüt kínálja föl, amíg a felhasználó a n° 9 opciót nem választja.
- 27. és 28. sorok : Az **adatRogzites()** osztály bármelyik adattábla bejegyzéseit elfogadja kezelésre. Egy szótárat – ami a felhasználó választásától függően megadja, hogy melyik nevet fogadjuk el (n° 3 vagy n° 4 opció) - használunk annak meghatározására, hogy az objektum létrehozásakor melyik adattáblát kell használni.
- 29. - 31. sorok : Az adatRogzites-objektum **enter()** metódusa aszerint ad 0 vagy 1 visszatérési értéket, hogy a felhasználó a bejegyzések rögzítésének folytatását vagy leállítását választotta. Ennek az értéknek a vizsgálata teszi lehetővé a hurok megszakítását.
- 35. és 44. sorok : Az **executeReq()** metódus aszerint ad 0 vagy 1 visszatérési értéket, hogy a server elfogadta-e a kérést vagy sem. Tehát azt eldöntendő vizsgálhatjuk ezt az értéket, hogy ki kell-e írni az eredményt vagy sem.

Gyakorlatok :

16.2. Módosítsa az előző oldalakon leírt scriptet úgy, hogy egy kiegészítő táblát ad az adatbázishoz. Ez például lehetne egy « zenekar » tábla, amiben mindegyik bejegyzés tartalmazná a zenekar nevét, a karmester nevét a hangszerek számát.

16.3. Adjon egy mezőt az egyik táblához (például egy ***float*** (valós) típusú vagy egy ***dátum*** típusú mezőt) és módosítsa a scriptet.

17. Fejezet : Webalkalmazások

Az olvasó biztosan sok mindent tanult már máshol a weblapok szerkesztéséről. Tudja, hogy ezek a lapok HTML formátumú dokumentumok, amiket egy hálózaton (intraneten vagy interneten) *webbrowsernek* vagy *navigátornak* nevezett programokkal (Netscape, Konqueror, Internet explorer, ...) lehet megnézni.

A HTML lapok egy másik számítógép - amin állandóan fut egy *Web servernek* (Apache, IIS, Zope, ...) nevezett alkalmazás - public könyvtárába vannak telepítve. Amikor létrejött egy kapcsolat a mi számítógépünk és e között a számítógép között, akkor a navigátorprogramunk párbeszédet folytathat a serverprogrammal (egy sor hardvereszköz és program közvetítésével, amikről most nem fogunk beszélni : telefonvonalak, routerek, cache-ek, kommunikációs protokolok ...).

A weblapok átvitelét kezelő HTTP protokol kétirányú adatcserét engedélyez. Azonban az esetek nagy többségében az információ átvitel gyakorlatilag csak egyirányú, tudni illik a server felől a navigátor felé : szövegeket, képeket, különféle fileokat küld neki nagy számban (ezek a weblapok, amiket megnézünk); ezzel szemben a navigátor a servernek csak kevés információt küld : lényegében azoknak a lapoknak az URL-ét, amiket a szörföző meg akar nézni.

17.1 Interaktív weblapok

Tudjuk viszont, hogy vannak olyan weblapok, amik arra szólítanak fel, hogy szolgáltassunk nagyobb mennyiségű információt : adjunk meg személyünkre vonatkozó adatokat egy klubba való beíratkozáskor, vagy egy szállodai szobafoglaláskor, a hitelkártyaszámunkat egy e-kereskedelmi siteon egy termék megrendelésekor, véleményünket vagy javaslatainkat, stb.

Sejtjük, hogy egy ilyen esetben, mint az említettek egyike, az átvitt információt a serveroldalon egy speciális programnak kell kezelni. Tehát az ilyen, információ fogadására szánt weblapok el kell legyenek látva egy olyan mechanizmussal, ami biztosítja az adatok átvitelét az információ kezelésére szánt program felé. Arra is szükség van, hogy ez a program információt tudjon átadni a servernek, hogy az a művelet eredményét egy új weblap formájában tudja prezentálni a szörfölőnek.

Ennek a fejezetnek az a célja, hogy megmagyarázza, hogyan használhatjuk fel Python programozási tudásunkat arra, hogy egy websitehoz - valódi alkalmazások beillesztésével - ilyen interakivitást adjunk.

Fontos megjegyzés : A következő fejezetben kifejtettek az olvasó iskolájának vagy vállalatának intranet-én azonnal működőképesek lesznek (feltétve, hogy az intranet adminisztrátora a servert megfelelő módon konfigurálta). Ami az internetet illeti, a helyzet egy kicsit bonyolultabb. Magától értetődik, hogy csak a rendszergazda beleegyezésével lehet programokat telepíteni egy internetre kapcsolt serverre. Ha egy internetellátó bizonyos diszkterületet bocsátott a rendelkezésünkre, ahol engedélyezve van « statikus » weblapok telepítése, az nem jelenti azt, hogy ott Python scripteket működtethetünk. Ahhoz, hogy az utóbbiak működhessenek, az internetellátónktól engedélyt és információkat kell kérnünk. Meg kell kérdezni, hogy aktiválhatunk-e Pythonban írt CGI scripteket a weblapjainkról és mely könyvtár(ak)ba telepíthetjük őket.

17.2 A CGI interface

A CGI (*Common Gateway Interface*) interface a legtöbb webservernek az egyik komponense. Ez egy átjáró, ami lehetővé teszi a kommunikációt az ugyanazon a számítógépen futó más programokkal. CGI-vel más nyelveken (Perl, C, Tcl, Python ...) írhatunk scripteket.

Ahelyett, hogy a web-et előre megírt dokumentumokra korlátoznánk, a CGI lehetővé teszi, hogy a szörfölő navigátora segítségével megadott adatoktól függően hozzunk létre weblapokat. A CGI scriptekkel az alkalmazások széles skálája hozható létre : online regisztrációs szolgáltatások, adatbázis kereső eszközök, közvélemény kutatások, játékok, stb.

A CGI programozás tanítása egész kézikönyvek tárgyát képezheti. Ebben a kezdők számára írt könyvben csak néhány alapelvet magyarázok el, hogy összehasonlítás révén megértessem azt az óriási előnyt, amit olyan specializált alkalmazásserver modulok nyújtanak egy interaktív websiteot fejleszteni kívánó programozónak, mint a Karrigell, a CherryPy vagy a Zope.

17.2.1 Egy alap CGI interakció

A következők megértése érdekében feltételezem, hogy az olvasó iskolai vagy vállalati hálózatának adminisztrátora úgy konfigurált egy intranet webservert, hogy az olvasó egy saját könyvtárba telepíthessen HTML oldalakat és Python scripteket .

Első példánk egy rendkívül egyszerű weblap lesz. Egyetlen interaktív elemet teszünk rá, egy gombot. Ezzel a gombbal fogjuk elindítani azt a programot, amit majd a későbbiekben írok le .

Kódoljuk az alábbi HTML dokumentumot egy tetszőleges editorral (a sorszámokat ne írjuk be, mert csak a magyarázat megkönnyítése miatt vannak ott) :

```
1.   <HTML>
2.   <HEAD><TITLE> Pythonos gyakorlat</TITLE></HEAD>
3.   <BODY>
4.
5.   <DIV ALIGN="center">
6.   <IMG SRC="penguin.gif">
7.   <H2>Interaktív weblap</H2>
8.   <P>Ez a lap egy Python scripttel van összekapcsolva</P>
9.
10.  <FORM ACTION="http://Serveur/cgi-bin/input_query.py" METHOD="post">
11.  <INPUT TYPE="submit" NAME="send" VALUE="A script végrehajtása">
12.  </FORM>
13.
14.  </DIV></BODY></HTML>
```

Azt már biztosan tudja az olvasó, hogy a kezdő és a megfelelő záró <HTML>, <HEAD>, <TITLE>, <BODY> tag-ek minden HTML dokumentumban közösek. Ezért nem fogunk foglalkozni a szerepükkel.

Az 5. sorban alkalmazott <DIV> tag szokás szerint a HTML dokumentum külön részekre bontására szolgál. Itt arra használom, hogy egy olyan részt definiálok, amiben minden elem (vízszintesen) középre van igazítva.

A 6. sorba beszúrok egy képet.

A 7. sor másodlagos címként egy szövegsort definiál.

A 8. sor egy szokványos bekezdés.

A 10.-12. sorok tartalmazzák a lényeges kódot (amivel foglalkozunk). A <FORM> és </FORM> tag-ek egy formot definiálnak, vagyis a weblap egy része különböző widgeteket tartalmazhat - adatbeviteli mezőket, gombokat, jelölőnégyzeteket, rádiógombokat, stb. -amikkel a szörfölő bizonyos tevékenységeket végezhet.

A <FORM> tag-nek két fontos paramétert kell tartalmazni : a form elküldésekor végrehajtandó **akciót** (itt annak a programnak az URL-ét kell megadni, amit az átvitt adatok kezeléshez kell hívni) és az információ átvitelére használt **módszert** (ami minket illet, ez mindíg a « post » módszer lesz).

Példánkban a hívni kívánt program egy **input_query.py** nevű Python script, ami az intranet server egy speciális könyvtárában van. Sok szerveren ennek a könyvtárnak a neve *cgi-bin.* Ez csak egy konvenció. Itt azt feltételezem, hogy az iskolai intranet adminisztrátora a Python scripteknek ugyanabba a könyvtárába való telepítését engedélyezte, mint amelyikben az olvasó saját weblapjai vannak.

Ezért a példánk 10. sorában lévő *http://Serveur/cgi-bin/input_query.py* címet azzal a címmel kell helyettesíteni, amit az adminisztrátor megad[67].

A 11. sor tartalmazza azt a tag-et, ami egy « elküldés gomb » (<INPUT TYPE="submit"> tag) típusú widgetet definiál. A VALUE ="texte" attribútummal adjuk meg azt a szöveget, aminek a gombon kell megjelenni. Jelen esetben a NAME paraméter megadása nem kötelező. Ez a widget nevét jelöli (arra az esetre, ha a célprogramnak szüksége lenne rá).

Amikor befejeztük a kódolást, mentsük a dokumentumot tetszőleges néven – .html vagy .htm filenév kiterjesztéssel - abba a könyvtárba, ami a weblapjaink elhelyezésére szolgál (például : essai.html).

Az **input_query.py** Python scriptet az alábbiakban részletezem. Mint már föntebb említettem, a scriptet ugyanabba a könyvtárba lehet telepíteni, mint amiben a HTML dokumentumunk van. :

```
1.    #! /usr/bin/python
2.
3.    # Egy egyszerűsített HTML űrlap kiíratása :
4.    print "Content-Type: text/html\n"
5.    print """
6.    <H3><FONT COLOR="Royal blue">
7.    Python scripttel elõállított weblap
8.    </FONT></H3>
9.
10.   <FORM ACTION="print_result.py" METHOD="post">
11.   <P>Kérem írja be a nevét az alábbi mezőbe :</P>
12.   <P><INPUT NAME="latogato" SIZE=20 MAXLENGTH=20 TYPE="text"></P>
13.   <P>Írjon be egy tetszőleges mondatot is :</P>
14.   <TEXTAREA NAME="mondat" ROWS=2 COLS=50>Mississippi</TEXTAREA>
15.   <P>Ezt a mondatot fogom hisztogramkészítésre használni.</P>
16.   <INPUT TYPE="submit" NAME="send" VALUE="Action">
17.   </FORM>
18.   """
```

67 Például : http://192.168.0.100/cgi/Classe6A/Dupont/input_query.py .

A script semmi mást nem csinál, mint kiír egy új weblapot, ami megint egy form-ot tartalmaz, de az most kidolgozottabb, mint az előző volt.

Az első sor nélkülözhetetlen : ez jelzi a CGI interface-nek, hogy a script végrehajtásához el kell indítani a Python interpretert. A második sor egy komment.

A 4. sor nélkülözhetetlen. Ez teszi lehetővé a Python interpreternek egy valódi HTML dokumentum inicializálását, amit át fog adni a webserver-nek. Ez utóbbi vissza fogja küldeni a HTML dokumentumot a szörfölő navigátorának, ami kiíratja azt.

Az utána következő tiszta HTML kódot a Python egyszerű karakterláncként kezeli, amit a **print** utasítással íratunk ki. Hogy be tudjuk szúrni mindazt, amit szeretnénk – a soremeléseket, aposztrófokat, idézőjeleket is beleértve -, ezért ezt a stringet <<< >>> -lel határoljuk. (Emlékezzünk rá, hogy a HTML figyelmen kívül hagyja a soremeléseket, tehát annyit használhatunk belőlük a kódunk « levegősebbé » és így olvashatóbbá tételére, amennyit akarunk).

17.2.2 Egy adatgyűjtésre szolgáló HTML form

Elemezzük most a HTML kódot. Lényegében egy több bekezdést tartalmazó új formot találunk benne, amiben több widget ismerhető fel. A 10. sor megadja annak a CGI scriptnek a nevét, aminek a form adatai át lesznek adva : nyilván egy másik Python scriptről lesz szó.

A 12. sorban egy « adatbeviteli mező » (INPUT TYPE="text") típusú widget definícióját találjuk. Ez arra ösztönzi a felhasználót, hogy írja be a nevét. A MAXLENGTH paraméter a beírandó karakterlánc maximális hosszát határozza meg (esetünkben 20 karakter). A SIZE paraméter a mező méretét a képernyőn határozza meg. A NAME paraméter az a név, amit annak a változónak választunk, amiben a várt stringet fogjuk tárolni.

A 14. sorban egy második, kissé eltérő adatbeviteli mező van definiálva (TEXTAREA tag). Egy nagyobb mezőről van szó, ami többsoros szövegek fogadására szolgál. (Ez a mező automatikusan gondoskodik görgető sávokról, ha a bevitt szöveg túl terjedelmes). A ROWS és COLS paraméterek neve elég kifejező. A nyitó és záró tag-ek közé alapértelmezett szöveget írhatunk (példánkban : « Mississippi »).

Mint az előző példában, a 16. sor tartalmazza annak a gombnak a definícióját, amire rá kell kattintani, hogy átadjuk az adatokat a cél CGI scriptnek, amit a következőkben fogok leírni.

17.2.3 Egy adatkezelésre szolgáló CGI script

A CGI scriptben egy nagyon egyszerű mecanizmust használunk a HTML formmal átadott adatok fogadására. Az alábbi példában ezt elemezhetjük. :

```
1.    #! /usr/bin/python
2.    # HTML form-mal átadott adatok kezelése
3.
4.    import cgi                          # interface modul a webserverrel
5.    form = cgi.FieldStorage()          # A felhasználói kérés fogadása :
6.                                       # egyfajta szótárról van szó
7.    if form.has_key("phrase"):         # A kulcs nem létezik, ha a
8.       text = form["mondat"].value     # megfelelő mező üres maradt
9.    else:
10.      text ="*** a phrase mező üres volt ! ***"
11.
12.   if form.has_key("latogato"):       # A kulcs nem létezik, ha a
13.      nomv = form["latogato"].value   # megfelelő mező üres marad
14.   else:
15.      nomv ="Ön nem adta meg a nevét"
16.
17.   print "Content-Type: text/html\n"
18.   print """
19.   <H3>Köszönöm, %s !</H3>
20.   <H4>Ön a következő mondatot adta meg: </H4>
21.   <H3><FONT Color="red"> %s </FONT></H3>""" % (nomv, text)
22.
23.   histogr ={}
24.   for c in text:
25.      histogr[c] = histogr.get(c, 0) +1
26.
27.   liste = histogr.items()        # átalakítás tuple-k listájává
28.   liste.sort()                   # a lista rendezése
29.   print "<H4>A mondat karaktereinek előfordulási gyakoriságai :</H4>"
30.   for c, f in liste:
31.      print 'a <B>"%s"</B> karakter %s alkalommal fordul elő <BR>' % (c, f)
```

A 4. és 5 sorok a legfontosabbak.

A 4. sorban importált **cgi** modul biztosítja a Python script kapcsolatát a CGI interface-szel, ami lehetővé teszi a párbeszédet a webserver-rel.

Az 5. sorban a modul **FieldStorage()** függvénye visszatérési értékként egy objektumot ad, ami a HTML formmal átadott adatokat tartalmazza. A **form** nevű változóba tesszük ezt a szótárra nagyon hasonlító objektumot.

Egy valódi szótár és a **form** nevű változóba tett objektum között az a lényeges különbség, hogy az utóbbiból a **value()** metódussal kell kiszedni az értékeket. A szótárakra alkalmazható többi metódust, mint amilyen például a **has_key()**, a szokásos módon használhatjuk.

A **FieldStorage()** visszatérési értékeként megadott szótárobjektum fontos jellemzője, hogy a megfelelő HTML formban ***üresen hagyott mezők számára nincs kulcsa***.

Példánkban a formnak két adatbeviteli mezője van, amikhez a « latogato » és « mondat » neveket rendeltük. Ha a felhasználó kitöltötte őket, akkor a tartalmukat a szótárobjektumban a « latogato » és a « mondat » indexeken fogjuk megtalálni. Viszont, ha valamelyik mezőt nem töltöttük ki, akkor az annak megfelelő index nem fog létezni. Tehát az értékek bármiféle kezelése előtt feltétlenül meg kell bizonyosodni a várt indexek létezéséről. Ezt tesszük a 7-15 sorokban.

(17) Gyakorlat :

17.1. Az előzőek igazolásaként például kikommentezhetjük a script 7., 9., 10., 12., 14. & 15. sorait. Ha ellenőrizzük a működését, látni fogjuk, hogy minden rendben megy, ha a felhasználó az összes mezőt kitölti. Ha viszont az egyik mezőt üresen hagyja, akkor egy hiba keletkezik.

Fontos megjegyzés: mivel a scriptet egy weblap segítségével indítottuk el, ezért a Python hibaüzenetek nem lesznek kiírva erre a weblapra, hanem a webserver eseménynaplójába lesznek bejegyezve. Beszélje meg a server adminisztrátorával, hogy hogyan férhet hozzá ehhez a naplóhoz. Minden esetre számítson rá, hogy egy CGI scriptben sokkal nehezebb a hibakeresés, mint egy közönséges alkalmazásban.

A script többi része elég szokványos.

- A 17-21 sorokban csak a form-mal átadott adatokat írjuk ki. Jegyezzük meg, hogy a **nomv** és **text** változóknak már előzetesen létezniük kell, ami nélkülözhetetlenné teszi a 9., 10., 14. és 15. sorokat.
- A 23., 24. és 25. sorokban egy szótárat használunk hisztogramkészítésre, ahogyan azt a 149. oldalon magyaráztam.
- A 27. sorban a szótárat egy tuple-kből álló listává alakítjuk, hogy a 28. sorban az utóbbit névsor szerint tudjuk rendezni.
- A 30. és 31. sorok **for** ciklusához nem kell kommentár.

17.3 Egy webserver Pythonban !

Az előző oldalakon azért magyaráztam el néhány CGI programozási alapismeretet, hogy jobban megértsük hogyan működik egy webalkalmazás. Ha az olvasó valóban szeretne egy ilyen alkalmazást készíteni (például egy website-ot, ami interaktív), akkor rá fog jönni, hogy a CGI interface túlságosan kezdetleges eszköz. Túlságosan nehézkes a használata ezekben a scriptekben, ezért jobban kidolgozott eszközöket kell igénybevenni.

Rendkívül jelentőssé vált a webfejlesztés iránti érdeklődés és erős az igény az ehhez a feladathoz adaptált interface-ek és programozási környezetek iránt. Még ha nem is olyan univerzális, mint a C/C++, a Pythont már mindenüt széles körben alkalmazzák igényes programok írására, így az webserver alkalmazások területén is. A nyelv stabilitása és egyszerű kivitelezése számos tehetséges fejlesztőt vonzott, akik rendkívül magasszintű webfejlesztő eszközöket készítettek. Ezek közül az alkalmazások közül több is érdekelheti az olvasót, ha sajátmaga akar különböző típusú interaktív websiteokat készíteni.

A létező termékek többsége szabad szoftver. A szükségletek széles skáláját fedik le, a néhány oldalas kis személyes website-tól a nagyméretű, kollaboratív kereskedelmi site-ig, ami napi többezer kérésre képes válaszolni és aminek különböző részeit változatos ismeretekkel rendelkező személyek (grafikusok, programozók, adatbázis specialisták, stb.) kezelnek anélkül, hogy zavarnák egymást.

A leghíresebb ezek közül a termékek közül a **Zope**, amit már nagy magán és nyilvános szervezetek adaptáltak kollaboratív intranet és extranet fejlesztésekre. Egy nagyon versenyképes,

biztonságos, majdnem teljesen Pythonban írt alkalmazásserver-ről van szó, amit egy egyszerű webinterface segítségével távolról felügyelhetünk. A **Zope** alkalmazás leírására a téma túl terjedelmes volta miatt nem lenne elég egy könyv. De tudjunk róla, hogy ez a termék tökéletesen alkalmas nagyon nagy vállalati websiteok kezelésére, miközben rendkívüli előnyöket kínál a klasszikus PHP-s és Java-s megoldásokhoz képpest.

Más, kevésbé igényes, de úgyszintén érdekes eszközök is rendelkezésre állnak. A Zope-hoz hasonlóan a többségük szabadon letölthető az internetről. Egyébként az, hogy Pythonban vannak írva biztosítja portabilitásukat : ugyanúgy tudjuk őket Windows alatt használni, mint Linux vagy MacOs alatt. Mindegyikük használható olyan « klasszikus » webserverrel, mint az Apache vagy a Xitami (ez a preferálandó, ha a megvalósítandó site-ot nagyon nagy terhelésre szánjuk). Egyébként egyesek közülük saját webservert foglalnak magukba, ami lehetővé teszi, hogy teljesen autonóm módon működjenek. Ez a lehetőség különösen hasznos egy site készítésekor, mert egyszerűsíti a hibakeresést.

A site-ok elkészítésének egyszerűségével társulva ez a teljes autonómia az említett termékeket nagyon jó megoldásokká teszi - a kis és középvállalkozásoknál, az adminisztrációban vagy az iskolákban - specializált intranetsite-ok megvalósítására. Ha az olvasó olyan Python alkalmazást akar fejleszteni, aminek egy egyszerű navigátorral kell elérhetőnek lenni egy vállalati intraneten keresztül (vagy éppen az interneten keresztül, ha az előrelátható terhelés nem túl jelentős), akkor ezek az alkalmazások az olvasónak készültek.

Számos változatuk létezik : *Poor man's Zope, Spyce, Karrigell, Webware, Cherrypy, Quixote, Twisted,* stb. Válasszon igényeitől függően. A bőség zavarával fog küzdeni.

A következőkben egy ***Karrigell***-lel működő webalkalmazást fogok leírni. A rendszer a ***http://karrigell.sourceforge.net*** címen található. Ez egy angolul jól dokumentált, egyszerű webfejlesztés (Pierre Quentel a szerzője, a *karrigell* breton szó jelentése : taliga).

17.3.1 A Karrigell telepítése

A telepítés gyerekjáték : az internetről letöltött arhív filet ki kell csomagolni egy tetszőleges könyvtárba. A kicsomagolás automatikusan létrehoz egy **Karrigell-verziószám** nevű alkönyvtárat. Ezt a könyvtárat fogjuk a következőkben gyökérkönyvtárnak tekinteni.

Ha nem szándékozzuk a *Karrigell* kiegészítőjeként adott ***Gadfly***[68] adatbázisservert használni, akkor ez minden, amit tennünk kell ! Ha igen, akkor menjünk a **gadfly-1.0.0** alkönyvtárba és írjuk be a `python setup.py install` parancsot (Linux alatt *root*-nak kell lennünk). Ha a beépített demo-t teljes egészében meg akarjuk nézni, ezt az utasítást kell végrehajtani.

17.3.2 A server indítása :

Egy webservert indítunk el, amihez aztán lokálisan vagy a hálózaton át bármilyen navigátorral hozzáférhetünk. Az indítás előtt tanácsos egy pillantást vetni a gyökérkönyvtárban található **Karrigell.ini** konfigurációs file-ra.

Alapértelmezetten a *Karrigell* a 80-as porton vár *http* kéréseket. A navigátorok többsége ezt a portszámot használja alapértelmezetten. Ha azonban egy Linuxos gépen telepítjük a *Karrigell*-t, akkor (biztonsági okokból) nincs jogunk az 1024 alatti portok használatára. Ha ez a helyzet áll fönt, akkor módosítani kell a konfigurációs filet, hogy a *Karrigell* egy magasabb portszámot használjon. Általában a 39. sor előtti # karakter eltávolítását választjuk, ami a 8080 portot aktiválja. Később esetleg azért kívánjuk módosítani a konfigurációs filet, hogy megváltoztassuk a website-unk

68 Lásd az előző fejezetet : a Gadfly egy Pythonban írt adatbázisserver

gyökérkönyvtárát (alapértelmezetten ez a server könyvtára).

Ha már módosítottuk a konfigurációs filet, menjünk a server gyökérkönyvtárába, ha még nem volnánk ott és írjuk be a következő parancsot :

```
python Karrigell.py
```

Ez minden. A *Karrigell* server elindul és kedvenc navigátorunkkal ellenőrizhetjük a működését. Ha a navigátort ugyanazon a gépen indítjuk el, mint amin a server van, akkor egy ilyen címet kell megadnunk : *http://localhost:8080/index.html* A « localhost » a lokális gépet jelenti, « 8080 » a konfigurációs fileban választott portszám és az « index.html » a site home page-e. Ha viszont egy másik gépről akarunk hozzáférni ugyanehhez a home page-hez, a navigátorban a *localhost* helyén a server IP címét kell megadnunk.

Az előző szakaszban[69] megadott címmel a *Karrigell* egy demo site-ját érjük el, ami már telepítve van a gyökérkönyvtárba. Ott találjuk az alapdokumentációt és egy sor példát.

Az előzőekben hallgatólagosan feltételeztem, hogy a servert egy konzolszöveggel vagy egy terminálablakból indítottuk. A server üzenetei egyik esetben a konzolban, másik esetben az ablakban jelennek meg. Ott kereshetjük az esetleges hibaüzeneteket. Ott kell beavatkozni, ha le akarjuk állítani a servert (a CTRL-C billentyű kombinációval).

17.3.3 Egy websitevá z

Próbáljuk meg elkészíteni a saját website-vázunkat. Egy klasszikus webservertől eltérően a *Karrigell* nemcsak statikus HTML oldalakat (*.htm*, *.html*, *.gif*, *.jpg*, *.css* fileokat), hanem :

- Python scripteket (*.py* fileokat)
- *HTML-en belüli hibrid Python* scripteket (*.pih* fileokat)
- *Python-on belüli HTML* hibrid *scripteket* (*.hip* fileokat) is kezelhet.

Hagyjuk a hibrid scripteket, az olvasó maga is tanulmányozhatja a szintaxisukat (ami egyébként rendkívül egyszerű), ha belekezd egy weboldal fejlesztésébe (megkönnyíthetik az életet). A könyv korlátai között megelégszem a közönséges Python scriptekkel végzett néhány kísérlettel.

Mint a site minden más elemét (*.html*, *.gif*, *.jpeg*, stb. fileokat), ezeket a Python scripteket is a gyökérkönyvtárban[70] kell elhelyezni. Rögtön el lehet végezni egy elemi tesztet. Írjuk be a következő egysoros scriptet :

```
print "Bienvenue sur mon site web !"
```

Mentsük el ezt a scriptet **hello.py** néven a gyökérkönyvtárba, majd írjuk be a navigátorba a *http://localhost/hello.py* vagy a : *http://localhost/hello* (a *.py* kiterjesztés elhagyható) címet. Látnunk kell megjelenni az üzenetet. Ez azt jelenti, hogy a *Karrigell* környezetben a **print** utasítás kimenete a kliens navigátorba van kiírva.

Mivel a kiírás egy navigátorablakba történt, ezért minden HTML szintaxisú forrást felhasználhatunk, hogy egy meghatározott formázást kapjunk. A következő utasításokkal például

69 Ha meghagytuk az alapértelmezett portszámot (80), akkor fölösleges azt megismételni a címekben, mert ezt a portszámot használják alapértelmezetten a navigátorok. Egy másik konvenció, hogy egy website home page-e majdnem mindíg az *index.htm* vagy *index.html* nevű file. Amikor egy website-ot a home page-én kezdve akarunk meglátogatni, akkor általában elhagyhatjuk ezt a nevet a címből. A *Karrigell* betartja ezt a konvenciót így egy egyszerűsített címmel kapcsolódhatunk a home page-hez, mint amilyen a következő : *http://localhost:8080* vagy *http://localhost* (ha a portszám 80).

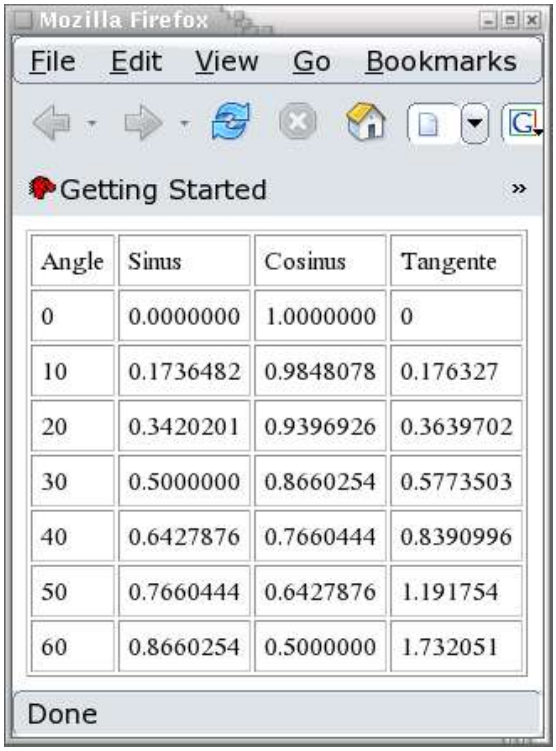
70 ...vagy a gyökérkönyvtár alkönyvtáraiban, ahogy az szokásos, amikor a konstrukció alatt lévő site megfelelő struktúrálására törekszönk. Ebben az esetben elég az alkönyvtár nevét bevenni a megfelelő címekbe.

kiírhatunk egy 2 sorból és 3 oszlopból álló táblázatot :

```
print """
<TABLE BORDER="1" CELLPADDING="5">
<TR> <TD> Rouge </TD> <TD> Vert </TD> <TD> Bleu </TD> </TR>
<TR> <TD> 15 % </TD> <TD> 62 % </TD> <TD> 23 % </TD> </TR>
</TABLE>
"""
```

Emlékezzünk rá, hogy a TABLE tag egy táblázatot definiál. A BORDER opciója az elválasztó keret szélességét, a CELLPADDING opciója a cellák tartalma körüli hézagot definiálja. A TR és TD (*Table Row* et *Table Data*) tag-ek a táblázat celláit és sorait definiálják

Természetesen minden Python erőforrást használhatunk, mint az alábbi példában, ahol a 0° és 90° közé eső szögek sinusaiból, cosinusaiból és tangenseiből készítünk táblázatot egy ciklus segítségével.



Angle	Sinus	Cosinus	Tangente
0	0.0000000	1.0000000	0
10	0.1736482	0.9848078	0.176327
20	0.3420201	0.9396926	0.3639702
30	0.5000000	0.8660254	0.5773503
40	0.6427876	0.7660444	0.8390996
50	0.7660444	0.6427876	1.191754
60	0.8660254	0.5000000	1.732051

7. sor : A **range()** függvényt használjuk a szögtartomány definiálására (0°-tól 60°-ig 10°-onként).

9. sor : A Python trigonometrikus függvényeinek radiánban kell a szöget megadni. Ezért egy átalakítást kell végezni.

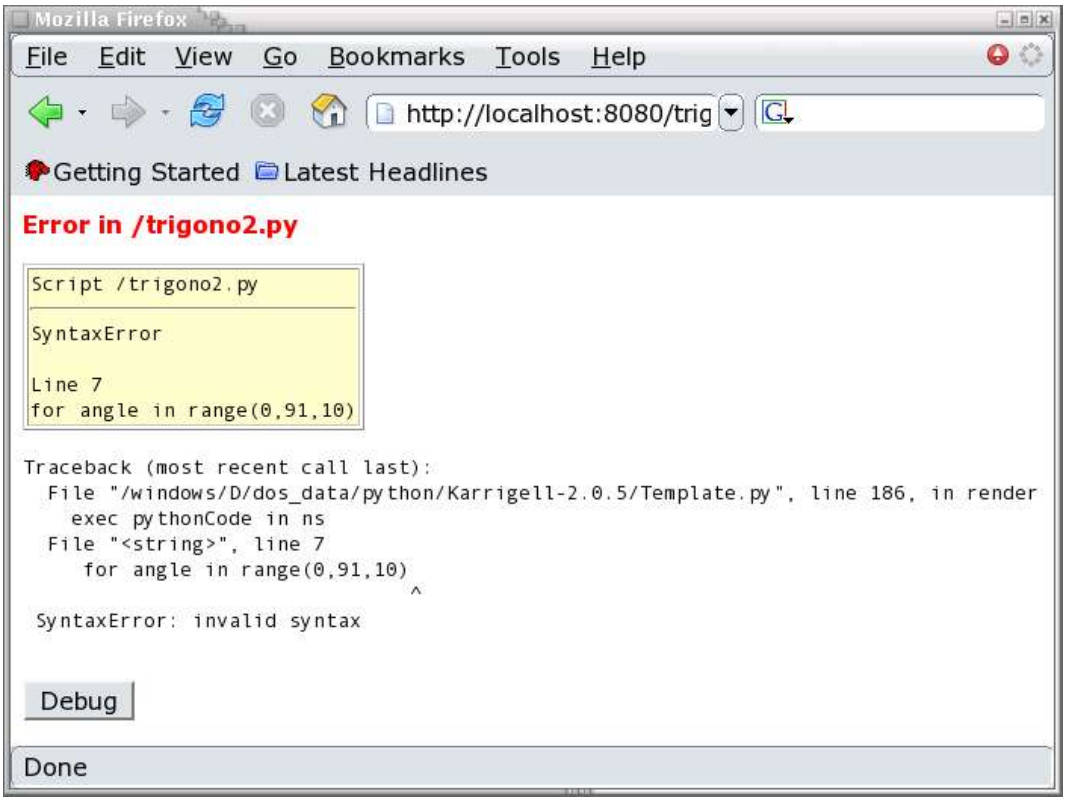
12. sor : A táblázat mindegyik sora 4 értéket tartalmaz. A 130. oldalon leírt stringformázó rendszer segítségével formázunk : a « %8.7f » konverziós marker 8 pozíción íratja ki az értéket, amiből 7 a tizedespont után van A « %8.7g » marker ugyanezt csinálja, csak tudományos formátumban, ha szükséges.

```
1.  from math import sin, cos, tan, pi
2.
3.  # Táblázatfej létrehozása oszlopcímekkel :
4.  print """<TABLE BORDER="1" CELLPADDING="5">
5.  <TR><TD>Angle</TD><TD>Sinus</TD><TD>Cosinus</TD><TD>Tangente</TD></TR>"""
6.
7.  for angle in range(0,62,10):
8.      # fokból radiánná alakítás :
9.      aRad = angle * pi / 180
10.     # a táblázat egy sorának létrehozása, kihasználjuk a stringformázást
11.     # a kiírás finomítására :
12.     print "<TR><TD>%s</TD><TD>%8.7f</TD><TD>%8.7f</TD><TD>%8.7g</TD></TR>"\
13.           % (angle, sin(aRad), cos(aRad), tan(aRad))
14.
15. print "</TABLE>"
```

Ebben a stádiumban az olvasó talán azt kérdezi magától, hogy mi a különbség aközött, amivel most kísérleteztünk és egy klasszikus CGI script között (mint amilyet a 267. és az azt követő oldalakon találunk).

Amikor hibázunk, nagyon gyorsan érzékelhetővé válik annak a jelentősége, hogy egy olyan speciálisabb környezetben dolgozunk, mint a *Karrigell*. A klasszikus CGI programozásban a Python interpreter által kiadott hibaüzenetek nem íródnak ki a navigátor ablakába. A server (például Apache) egy naplófile-jába vannak bejegyezve, ami nem egyszerűsíti a megtekintésüket.

Egy olyan eszközzel mint a *Karrigell* viszont egy nagyon hatékony jelzőrendszerrel, valamint egy komplett debugerrel rendelkezünk. Kísérletként csináljunk egy kis hibát a fenti scriptben és töltsük be újre a navigátorba a módosított weblapot. Például töröljük a kettőspontot a 7. sor végéről, ekkor a következő kiírást kapjuk :



A « Debug » gombra kattintva rengeteg kiegészítő információt kapunk (kiírja a komplet scriptet, a környezeti változókat, stb.).

17.3.4 Session-ök kezelése

Egy interaktív websitet készítésekor sokszor azt akarjuk, hogy a látogató tudja magát azonosítani és a különböző lapok látogatása során mindvégig tudjon információkat szolgáltatni (erre típuspélda a « bevásárlókocsi » feltöltése egy kereskedelmi site látogatása során) úgy, hogy ezek az információk megőrződnek valahol egészen a látogatás végéig. Természetesen ezt minden csatlakozott kliensre egymástól függetlenül kell megvalósítani.

Az információk lapról lapra történő átadása rejtett form-mezőkkel lehetséges lenne, de ez komplikált és nagyon korlátozó jellegű lenne. Kívánatos, hogy a servernek legyen egy speciális mechanizmusa, ami mindegyik klienshez hozzárendel egy specális ***session***-t. A *Karrigell* ezt a célt *cookie*-kkal éri el. Amikor egy új látogató azonosítja magát, a server létrehoz egy *sessionId*-nek nevezett *cookie*-t és elküldi a navigátornak, ami elmenti azt. Ez a *cookie* egy egyedi « session azonosítót » tartalmaz, aminek a serveren egy session-objektum felel meg. Amikor a látogató a site más oldalait járja be, a navigátora minden alkalommal elküldi a *cookie* tartalmát a servernek és a server a sessionazonosító alapján meg tudja találni a megfelelő session-objektumot. Így a session-objektum a szörfölő látogatása során végig rendelkezésre áll : egy közönséges Python objektumról van szó, amiben attribútumok formájában bármennyi információt tárolhatunk.

A programozás szintjén ez a következő képpen történik :
Minden oldalt, amin meg akarunk nézni vagy módosítani akarunk egy session információt, azzal

kezdünk, hogy létrehozzuk a **Session()** osztály egy objektumát :

```
objet_session = Session()
```

Ha a session elején vagyunk, akkor a *Karrigell* generál egy egyedi azonosítót, azt elhelyezi egy *cookie*-ban és a cookie-t elküldi a navigátornak. Tetszőleges számú attribútumot adhatunk a sessionobjektumhoz :

```
objet_session.nev = "Jean Dupont"
```

A többi lapon hasonló módon járunk el, de ebben az esetben a **Session()** osztályból létrehozott objektum nem egy új objektum, hanem a session elején létrehozott objektum, amit a server belsőleg a *cookie*-beli azonosító visszaolvasása révén talált meg. Az attribútumai értékeihez hozzáférhetünk és az objeltumhoz új attribútumokat is hozzáadhatunk :

```
obj_sess = Session()              # a cookie-val megadott objektum megkeresése
om = obj_sess.nev                 # létező attribútumérték megtalálása
obj_sess.cikkszam = 49137         # új attribútum hozzáadása
```

A session-objektumoknak van egy **close()** metódusuk is, ami a session-információ törlésére való. Nem kötelező a session-ök explicit zárása :a *Karrigell* minden esetre biztosítja, hogy sose legyen egyidőben 1000-nél több session : amikor eléri az 1000. session-t, törli a legrégibbeket.

Kidolgozott példa :

Mentsük az alábbi három kis scriptet a gyökérkönyvtárba. Az első a fentebb leírthoz hasonló HTML formot generál. Legyen a neve : **sessionTest1.py** :

```
1.  # Egy beíratkozási form kiírása :
2.
3.  print """
4.  <H3>Kérem, azonosítsa magát :</H3>
5.
6.  <FORM ACTION = "sessionTest2.py">
7.  Családnév : <INPUT NAME = "kliensCsaladnev"> <BR>
8.  Utónév : <INPUT NAME = "kliensUtonev"> <BR>
9.  Neme (ffi/no) : <INPUT NAME = "kliensNeme" SIZE ="1"> <BR>
10. <INPUT TYPE = "submit" VALUE = "OK">
11. </FORM>"""
```

A következő neve legyen **sessionTest2.py**. Ez az a script, amit a fönti form nyitó tag-jében a 6. sorban adtunk meg és ami akkor lesz hívva, amikor a felhasználó a 10. sorban elhelyezett gombra kattint. A felhasználó által a form különböző mezőibe beírt értékeket a *Karrigell*[71] QUERY környezeti változójában lévő « kérésszótár » közvetítésével fogja fogadni :

```
1.  obSess = Session()
2.
3.  obSess.nev = QUERY["kliensCsaladnev"]
4.  obSess.utonev = QUERY["kliensUtonev"]
5.  obSess.nem = QUERY["kliensNeme"]
6.
7.  if obSess.nem.upper() == "M":
8.      megszolitas ="Monsieur"
```

71 *A Karrigell* globális változókat helyez el, amiknek a neve nagybetűs azért, hogy elkerülje az esetleges konfliktusokat a mi változóneveinkkel. Ez ugyanazt a szerepet játssza, mint a cgi modul *FieldStorage()* függvénye. Részletesebb magyarázatért nézze meg a *Karrigell* dokumentációját.

```
9.  else:
10.     megszolitas ="Madame"
11. print "<H3> Üdvözlöm, %s %s </H3>" % (megszolitas, obSess.nem)
12. print "<HR>"
13. print """
14. <a href = "sessionTest3.py"> Folytatás...</a>"""
```

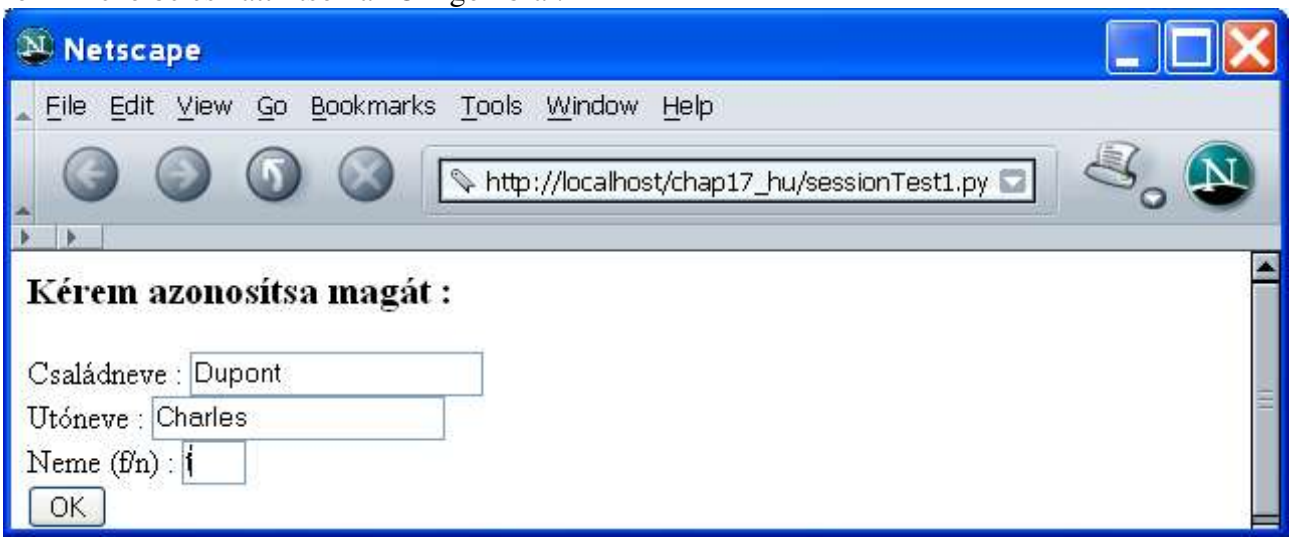
A script első sora létrehoz egy sessionobjektumot, generál neki egy egyedi azonosítót és *cookie* formájában elküldi a navigátornak.

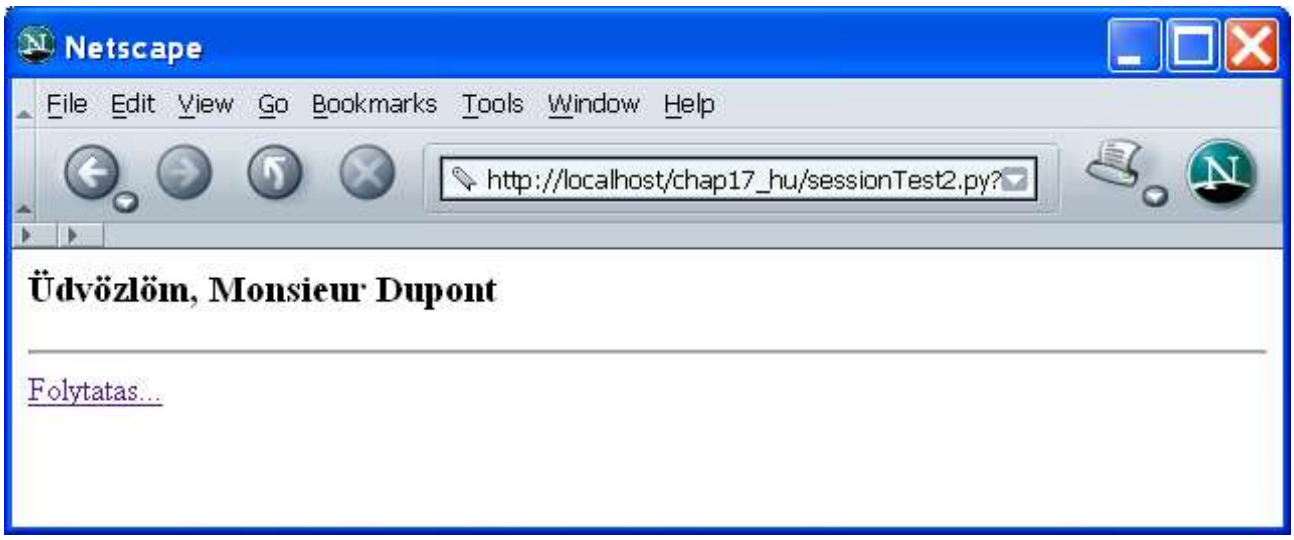
A 3,. 4., 5. sorokban úgy nyerjük ki az előző form mezőibe beírt értékeket, hogy a mezőneveket a « kérésszótár » kulcsaiként használjuk.

A 14. sor egy *http* linket definiál, ami a harmadik, **sessionTest3.py** nevű scriptre mutat :

```
1.  folytatasSess = Session()            # a session-objektum megtalálása
2.  folytatasSess.cikkszam = 12345       # attributumok hozzáadása
3.  folytatasSess.ar = 43.67
4.
5.  print """
6.  <H3> Következő lap </H3> <HR>
7.  Az ügyfél rendelésének követése : <BR> %s %s <BR>
8.  Article n° %s, Prix : %s <HR>
9.  """ % (folytatasSess.nev, folytatasSess.utonev,
10.        folytatasSess.cikkszam, folytatasSess.ar)
```

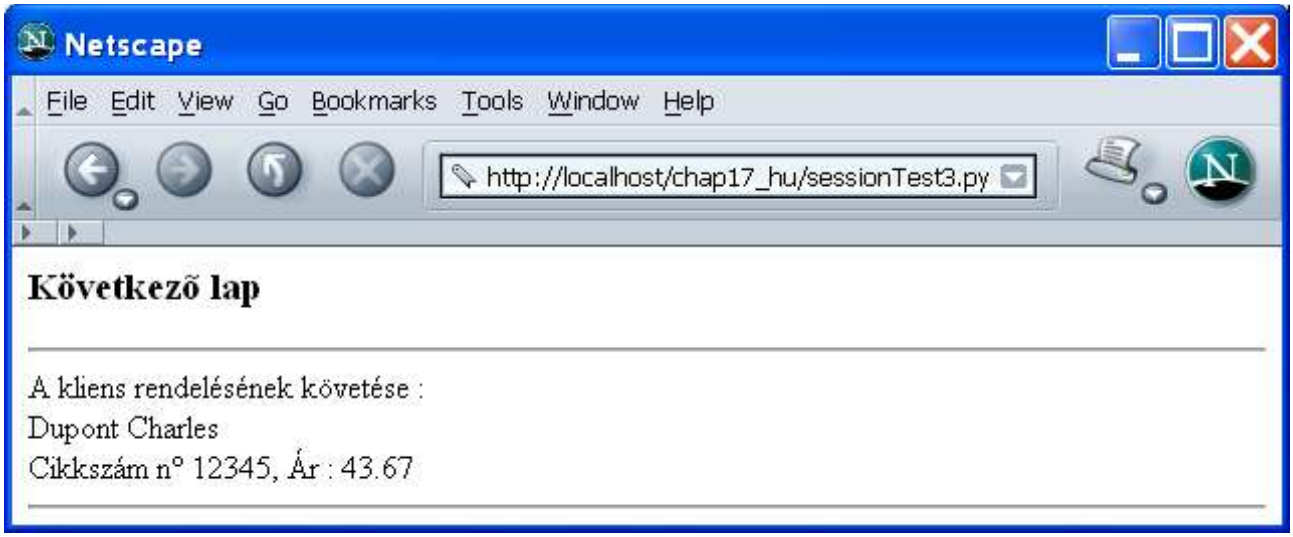
Írja be a navígátorba a *http://localhost:8080/sessionTest1 címet.* Írjon be tetszőleges értéket a form mezőibe és kattintson az OK gombra :





Ahogy vártuk, a formba beírt információk át lettek adva a második lapnak. Ha most a « Folytatás... » linkre kattintunk, akkor egy újabb lap töltődik be, de ide nem történik semmilyen adatátadás sem (mert nem form közvetítésével férünk hozzá). A **sessionTest3.py** scriptben, ami ezt a lapot generálja tehát nem használhatjuk a QUERY változót a látogató által beírt információk megkeresésére.

Itt lép be a session-objektum mechanizmus. A harmadik script indításakor a server elolvassa a navigátor által tárolt *cookie*-t, ami lehetővé teszi neki az előző scriptben létrehozott session-objektum újragenerálását.



Elemezzük a **sessionTest3.py** script első három sorát : a **Session() osztályból** létre hozott **folytatasSess** objektum a visszaállított session-objektum. Az előző oldalon elmentett információkat tartalmazza és kiegészítő attribútumokban más információkat adhatunk hozzá.

Innentől kezdve azonos módon szedhetjük össze az információt bármelyik másik oldalról, mert azok mindaddig meg fognak maradni, amíg a felhasználó be nem fejezi a sitelátogatást, kivéve ha a **close()** metódus hívásával programból zárjuk be ezt a sessiont.

Gyakorlat :

17.2. Adjon az előző scripthez egy negyedik lapra mutató linket és írjon egy scriptet, ami létrehozza azt. Az információkat ez esetben egy táblázatba kell írni :

Családnév	*Utónév*	*Nem*	*Árucikk*	*Ár*

17.3.5 Egyéb fejlesztések

Itt befejezzük a *Karrigell* rövid tanulmányozását, mert úgy tűnik elmondtam amit az elinduláshoz tudni kell. Ha az olvasó többet akar tudni, akkor meg kell nézni a dokumentációt és a termékkel adott példákat. Mint már föntebb jeleztem, a *Karrigell* telepítése tartalmazza a *Gadfly* adatbázisrendszer telepítését is. Nagyon gyorsan és könnyen készíthetünk olyan interaktív site-ot, amivel bármilyen adategyüttest megtekinthetünk, feltéve természetesen, hogy a site-unk terhelése mérsékelt lesz és az adatbázis nem válik gigantikus méretűvé. Ne reméljük, hogy a *Karrigell*-el napi több millió kérés kezelésére képes kereskedelmi site-ot fogunk tudni készíteni!

Ha ilyen ilyen ambícióink vannak, akkor más programokat kell megnéznünk, mint amilyen például az *Apache* serverrel összekapcsolt *CherryPy* vagy a *Zope* és adatbázis kezelőként az *SQLite*, *MySQL* vagy *PostgreSQL.*

18. Fejezet : Kommunikáció a hálózaton keresztül

Az internet rendkívüli mértékű fejlődése megmutatta, hogy a számítógépek nagyon hatékony kommunikációs eszközök lehetnek. Ebben a fejezetben két program összekapcsolásának a legegyszerűbb technikájával fogunk kísérletezni, ami lehetővé teszi közöttük a hálózaton át történő információcserét.

A következőkben feltételezem, hogy az olvasó együttműködik egy vagy több osztálytársával és a Python munkaállomásaik egy TCP/IP kommunikációs protokollt használó lokális hálózatra csatlakoznak. Az operációs rendszernek nincs jelentősége. A későbbiekben leírt Python scriptek egyikét például egy Linux-szal működő munkaállomásra lehet telepíteni, és párbeszédre lehet késztetni egy másik scripttel, amit egy másik operációs rendszerrel - mint amilyen a MacOS vagy Windows - működő gépen futtatunk.

Az olvasó olyan kísérletet is végezhet, hogy megnézi mi történik, ha ugyanazon a gépen egymástól független ablakokban hajtatja végre a különböző scripteket.

18.1 A socket-ek

Az első gyakorlat, amit javaslok, csak két gép közötti kommunikáció létrehozásából áll. A két gép egymás után fog tudni üzenetet váltani, azonban az olvasó meg fogja tudni állapítani, hogy a konfigurációik nem szimmetrikusak. Az egyik gépre telepített script a ***serverprogram*** szerepét fogja játszani, míg a másik ***kliensprogramként*** fog viselkedni.

A serverprogram folyamatosan fut az egyik gépen, amelyiket egy specifikus ***IP cím*** azonosít a hálózaton[72]. Egy meghatározott ***kommunikációs porton*** állandóan figyeli azoknak a kéréseknek a beérkezését, amiket a potenciális kliensek e felé a cím felé küldenek. Ehhez a megfelelő scriptnek egy ***socket*** -nek nevezett programobjektumot kell létrehozni, ami össze van kapcsolva ezzel a porttal.

Egy másik gépről kiindulva a kliensprogram megfelelő kérést kibocsátva megkísérli létrehozni a kapcsolatot. Ez a kérés egy hálózatra bízott üzenet, ami olyan, mint egy levél, amit a postára bízunk. A hálózat bármelyik másik gép felé elindíthatná a kérést, de csak egyetlen egy gép van megcélozva : hogy a rendeltetési helyét elérhesse, a kérés header-je tartalmazza a célgép IP címét és kommunikációs portját.

Amikor létrejött a serverrel a kapcsolat, a kliens hozzárendeli saját magához az egyik kommunikációs portját. Ettől a pillanattól kezdve úgy tekinthetjük, hogy egy privilegizált csatorna köti össze a két gépet úgy, mintha egy kábel kötné őket össze (a két kommunikációs port játsza a kábel két végének a szerepét). Az információcsere elkezdődhet.

Hogy a hálózati kommunikációs portokat használni tudják, a programok a ***socket*** -eknek nevezett interface objektumok közvetítésével az operációs rendszer eljárásainak és függvényeinek egy csoportját hívják. Ezek két különböző és komplementer technikát alkalmazhatnak : az interneten széles körben alkalmazott packet (ezeket datagrammoknak is nevezik) technikát és a folytonos kapcsolat vagy ***stream socket*** technikát, ami egy kicsit egyszerűbb.

72 Egy adott gépet egy explicit névvel is megadhatunk azzal a feltétellel, hogy a hálózaton telepítve van egy olyan mecanizmus (DNS), ami ezt a nevet automatikusan IP címre fordítja. Ha többet akar tudni erről, akkor nézzen utána az operációs rendszerek és hálózatok tananyagban.

18.2 Egy elemi server készítése

Első kísérletként a ***stream socket*** technikát fogjuk használni. Ez tökéletesen megfelel, ha , lokális hálózattal összekötött gépeknek kell egymással kommunikálni. Ezt a technikát nagyon könnyű alkalmazni és nagyobb adatátviteli sebességet tesz lehetővé.

A másik technológia (a package technológia) az interneten keresztül történő kommunikációknál lesz kívánatos a nagyobb megbízhatósága miatt (ugyanazok a csomagok különböző utakon érhetik el a rendeltetési helyüket, több példányban lehetnek kiküldve ha az adatátviteli hibák javítása miatt az szükségesnek látszik), de a kivitelezése összetettebb. Ezt a technológiát nem fogjuk ezen a kurzuson tanulni.

Az alábbi script egy olyan servert állít be, ami egyetlen klienssel tud kommunikálni. Kicsit később majd meglátjuk mivel kell kiegészíteni, hogy párhuzamosan több klienst tudjon kezelni.

```
1.   # Egy elemi hálózati server definíciója
2.   # Ez a server várja a kapcsolatot a klienssel, hogy párbeszédet kezdjen vele
3.
4.   import socket, sys
5.
6.   HOST = '192.168.14.152'
7.   PORT = 50000
8.
9.   # 1) A socket létrehozása:
10.  mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11.
12.  # 2) a socket összekapcsolása egy meghatározott címmel :
13.  try:
14.      mySocket.bind((HOST, PORT))
15.  except socket.error:
16.      print "A socket összekapcsolása a választott címmel meghiúsult."
17.      sys.exit()
18.
19.  while 1:
20.      # 3) Várakozás a kliens kapcsolatkérésére :
21.      print "A server kész, a kérésre vár ..."
22.      mySocket.listen(5)
23.
24.      # 4) A kapcsolat létrehozása :
25.      connexion, adresse = mySocket.accept()
26.      print "A kliens felkapcsolódott, IP cím %s, port %s" % (adresse[0], adresse[1])
27.
28.      # 5) Párbeszéd a klienssel :
29.      connexion.send("Ön a Marcel serverre kapcsolódott. Küldje el az üzenetét.")
30.      msgClient = connexion.recv(1024)
31.      while 1:
32.          print "C>", msgClient
33.          if msgClient.upper() == "FIN" or msgClient =="":
34.              break
35.          msgServeur = raw_input("S> ")
36.          connexion.send(msgServeur)
37.          msgClient = connexion.recv(1024)
38.
39.      # 6) A kapcsolat zárása :
40.      connexion.send("Viszontlátásra !")
41.      print "A kapcsolat megszakadt."
42.      connexion.close()
43.
44.      ch = raw_input("<U>jrakezdeni <B>efejezni ? ")
45.      if ch.upper() =='B':
46.          break
```

Magyarázatok :

- 4. sor : A **socket** modul minden, a kommunikációs programok készítéséhez szükséges függvényt és osztályt tartalmaz. A következő sorokban látni fogjuk, hogy a kommunikáció létrehozása hat lépésből áll.
- 6. és 7. sor : Ez a két változó definiálja a server-azonosítót úgy, ahogyan azokat a socket-be be fogjuk építeni. A HOST-nak a server IP címét megadó stringet kell tartalmazni a megszokott decimális formában, vagy a server DNS nevét (azzal a feltétellel, hogy a hálózaton egy névfeloldó mechanizmus van telepítve). A PORT -nak egy egész számot kell tartalmazni, tudni illik egy még használatlan port számát, ami lehetőség szerint 1024-nél nagyobb. (lásd a hálózati szolgáltatások kurzust).
- 9. és 10. sor : Az összekapcsolás első lépése. Létrehozzuk a **socket()** osztály egy objektumát. Két opciót adunk meg, amik megadják a választott címek típusát (« internet » típusú címeket fogunk használni) és az adatátviteli technológiát (datagrammok vagy folyamatos kapcsolat (*stream)* : ez utóbbi használata mellett döntöttünk).
- 12. - 17. sorok : Második lépés. Megkíséreljük létrehozni a kapcsolatot a socket és a kommunikációs port között. Ha nem hozható létre ez a kapcsolat (például a kommunikációs port foglalt vagy a gép neve inkorrekt), akkor a program egy hibaüzenettel befejeződik.
  Megjegyzés : a socket **bind()** metódusa egy tuple típusú argumentumot vár, emiatt kell a két változót két pár zárójelbe tenni.
- 19. sor : A serverprogramot egy végtelen ciklusban indítjuk el, mert állandóan várnia kell a potenciális klienseinek a kéréseit.
- 20. - 22. sorok : Harmadik lépés. Mivel a socket hozzá van kapcsolva egy kommunikációs porthoz, most felkészülhet a kliensek által küldött kérések fogadására. Ez a **listen()** metódus szerepe. Az argumentuma megmondja, hogy maximum hány párhuzamos kapcsolatot fogadjon el. Majd később meglátjuk hogyan kell ezeket kezelni.
- 24. - 26. sorok : Negyedik lépés. Amikor az **accept()** metódusát hívjuk, a socket mindaddig vár, amíg egy kérés megjelenik. A script ezen a részen megszakad, ez olyan, mintha az **input()** függvényt hívnánk, hogy a klaviatúráról várjunk bemenetet. Ha fogad egy kérést, akkor az **accept()** metódus visszatérési értékként egy kételemű tuple-t ad meg : az elsö elem a **socket()**[73] osztály egy új objektumának a hívatkozása, ami a kliens és a server közötti valódi kommunikációs interface lesz, a második elem egy másik tuple lesz, ami ennek a kliensnek az adatait tartalmazza (az IP címét és a portszámot, amit használ).
- 28. - 30 : Ötödik lépés. A kommunikáció létrejött. A socket **send()** és **recv()** metódusai nyilván az üzenetek – amiknek egyszerű stringeknek kell lenni - küldésére és fogadására szolgálnak.

  Megjegyzések : a **send()** metódus az elküldött byteok számát adja visszatérési értékként. A **recv()** metódusnak hívásakor argumentumként meg kell adni, hogy maximálisan hány byte-ot fogadjon egyszerre (A szám feletti byteok egy pufferben lesznek elhelyezve. A **recv**() metódus újabb hívásakor kerül sor az átvitelükre).

73 A későbbiekben majd meglátjuk, hogy mi a haszna annak, hogy inkább így hozunk létre egy új socket-objektumot a kommunikáció kezelésére, mint hogy a 10. sorban már létrehozott socket-objektumot használnánk. Röviden, ha azt akarjuk, hogy a serverünk több kliens kapcsolatát tudja szimultán kezelni, akkor mindegyik számára egy külön socket-tel kell rendelkeznünk az elsőtől függetlenül, amit állandóan működni hagyunk, hogy fogadja az új kliensektől származó kéréseket.

- 31. - 37. sorok : Ez az új végtelen ciklus tartja fenn a párbeszédet mindaddig, míg a kliens úgy nem dönt, hogy a « fin » szót, vagy egy üres stringet nem küld. A két gép monitorára ki lesz írva ennek a dialógusnak az előrehaladása.
- 39. - 42. sorok : Hatodik lépés. A kapcsolat zárása.

18.3 Egy elemi kliens konstruálása

Az alábbi script az előző oldalakon leírt serverprogram kiegészítő kliensprogramja. Látni fogjuk, hogy rendkívül egyszerű.

```
1.  # Egy elemi hálózati kliens definíciója
2.  # A kliens egy ad hoc serverrel folytat párbeszédet
3.
4.  import socket, sys
5.
6.  HOST = '192.168.14.152'
7.  PORT = 50000
8.
9.  # 1) socket létrehozása :
10. mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11.
12. # 2) kapcsolatkérés küldése a serverhez :
13. try:
14.     mySocket.connect((HOST, PORT))
15. except socket.error:
16.     print "A kapcsolat meghiúsult."
17.     sys.exit()
18. print "A kapcsolat létrejött a serverrel."
19.
20. # 3) Párbeszéd a serverrel :
21. msgServeur = mySocket.recv(1024)
22.
23. while 1:
24.     if msgServeur.upper() == "FIN" or msgServeur =="":
25.         break
26.     print "S>", msgServeur
27.     msgClient = raw_input("C> ")
28.     mySocket.send(msgClient)
29.     msgServeur = mySocket.recv(1024)
30.
31. # 4) A kapcsolat zárása :
32. print " A kapcsolat megszakítva."
33. mySocket.close()
```

Magyarázatok :

- A script eleje hasonlít a serverscript elejéhez. Az IP cím és a kommunikációs portszám a serveré kell, hogy legyen.
- 12. - 18. sorok : Most csak egyetlen socket-objektumot hozunk létre, aminek a **connect()** metódusát használjuk a kapcsolatkérés elküldésére.
- 20. - 33. sorok : Ha a kapcsolat létrejött, akkor a fentebb már leírt **send()** és **recv()** metódusokat használva kommunikálhatunk a serverrel.

18.4 Több párhuzamos task kezelése threadek (szálak) segítségével

Az előző oldalakon kidolgozott kommunikációs rendszer valóban nagyon elemi : egyrészt csak két gépet kapcsol össze, másrészt korlátozza a két beszélő szabadságát. Csak egymás után küldhetnek üzeneteket. Például amikor az egyikük kibocsátott egy üzenetet, a rendszere blokkolva marad addig, amíg a partnere nem küld neki választ. Amikor fogadott egy ilyen választ, akkor nem lesz képes egy másik válasz fogadására mindaddig, amíg maga nem küldött el egy új üzenetet, ... és így tovább.

Ezek a problémák onnan származnak, hogy a szokásos scriptjeink egyszerre csak egy dologgal tudnak foglalkozni. Amikor például az utasításfolyamban egy **input()** függvényre kerül a sor, mindaddig semmi sem fog történni, míg a felhasználó be nem írja a várt adatot. Még ha ez a várakozás nagyon hosszú is, rendszerint akkor sem lehetséges, hogy a program ez alatt az idő alatt más feladatot végezzen. Ez azonban csak egy és ugyanazon programon belül igaz : azt biztosan tudja az olvasó, hogy a számítógépen közben végre lehet hajtani más alkalmazásokat, mert a modern operációs rendszerek « multitask »-ok.

A következő oldalakat annak a magyarázatára szánom, hogy hogyan kell implementálni a programokban a multitask funkcionalitást, hogy párhuzamosan több partnerrel kommunikálni képes hálózati alkalmazásokat tudjunk fejleszteni.

Nézzük most meg az előző oldal scriptjét. A működésének a lényege a 23.-29. sorokban található programhurok. Ez a hurok két helyen megszakad :

- a 27. sorban, hogy megvárja a felhasználó adatbevitelét a billentyűzetről (raw_input() függvény)
- a 29. sorban, hogy egy hálózati üzenet beérkezését várja.

Ez a két várakozás ***egymás után*** következik, holott jóval előnyösebb lenne, ha ezek ***egyidejűek*** lennének. Ha ez lenne a helyzet, akkor a felhasználó minden pillanatban üzeneteket küldhetne anélkül, hogy minden esetben meg kellene várnia a partnere reakcióját. Akármennyi üzenetet fogadhatna anélkül, hogy kötelező lenne mindegyikre válaszolni, hogy más üzeneteket fogadhasson.

Erre az eredményre juthatunk, ha megtanuljuk hogyan kezelünk ***párhuzamosan*** több utasítássort ugyanabban a programban. De hogyan lehetséges ez ?

Az informatika története során többféle technikát kitaláltak arra, hogy megosszák egy processzor munkaidejét különböző programok között úgy, hogy úgy tűnjön azok egyidőben működnek (míg a valóságban a processzor mindegyikükkel egy kis ideig foglalkozik, amikor rájuk kerül a sor). Ezek a technikák implementálva vannak az operációs rendszerben. Nem szükséges őket itt részletezni még akkor sem, ha mindegyikük hozzáférhető a Pythonnal.

A következő oldalakon ezen technikák közül annak a használatát fogjuk megtanulni, ami egyszerre a legegyszerűbben kivitelezhető és az egyetlen valóban portábilis technika (lényegében minden nagy operációs rendszer támogatja) : a technikát ***light process***-nek vagy ***thread***[74]-nek nevezik.

Egy számítógépen a ***thread***-ek párhuzamosan (kvázi-egyidejűleg) kezelt utasítássorozatok, amik mind ugyanazon a globális névtéren osztoznak.

Valójában bármely Python program utasításai mindíg legalább egy thread-et követnek : a ***fő-***

74 Egy Unix típusú operációs rendszerben (mint amilyen a Linux), ugyanannak a programnak a különböző *thread*-jei egyetlen *process*nek képezik a részét. Ugyanannak a Python scriptnek a segítségével különböző processek kezelése is lehetséges (*fork* művelet), azonban ez a technika messze meghaladja ennek a kurzusnak a kereteit.

threadet. Ebből más « gyermek » thread-eket lehet elindítani, amik párhuzamosan lesznek végrehajtva. Amikor valamennyi utasítását végrehajtotta a gyermek process, akkor befejeződik és minden külön bejelentés nélkül eltűnik. Amikor viszont a fő-thread fejeződik be, néha meg kell róla győződni, hogy minden gyermek-threadje « meghal » vele.

18.5 Egyidejű küldést és fogadást kezelő kliens

Egy egyszerűsített « chat »[75]-rendszer készítéséhez fogjuk most a gyakorlatba átültetni a thread-technikát. Ez a rendszer egyetlen szerverből és tetszőleges számú kliensből fog állni. Szemben azzal , ami az első gyakorlatunkban történt, magát a szervert senki sem fogja kommunikációra használni, hanem amikor elindítjuk, több kliens kapcsolódhat majd rá és kezdhet el egymással üzenetet váltani.

Mindegyik kliens az összes üzenetét el fogja küldeni a server-nek, az pedig rögtön továbbküldi azokat az összes rákapcsolódott kliensnek úgy, hogy mindegyik láthassa a forgalom egészét. Mindegyik bármikor, bármilyen sorrendben elküldheti az üzeneteit és fogadhatja a többiekét, mert a fogadás és küldés egyidejűleg, külön thread-ekben van kezelve.

A következő script a kliensprogramot definiálja. A servert egy kicsit később fogom leírni. Megállapíthatjuk, hogy a script főrésze (a 38. és az azt követő sorok) hasonlít az előző példa scriptjének főrészéhez. Csak a « Párbeszéd a serverrel » részt helyettesítettem. A while hurok helyett most két thread-objektumot létrehozó utasítást (49. és 50. sor) találunk, amiknek a működését a következő két sorban indítjuk el. Ezeket a thread-objektumokat a **threading** modul **Thread()** osztályából leszármaztatással hozzuk létre. Egymástól függetlenül foglalkoznak az üzenetek fogadásával és kibocsátásával. A két « gyermek »-thread így tökéletesen be van zárva különböző objektumokba, ami megkönnyíti a mechanizmus megértését.

```
1.  # Üzenetek párhuzamos kibocsátását és fogadását kezelő
2.  # hálózati kliens definiálása (2 THREAD alkalmazása).
3.
4.  host = '192.168.0.235'
5.  port = 40000
6.
7.  import socket, sys, threading
8.
9.  class ThreadReception(threading.Thread):
10.     """üzenetek fogadását kezelő thread objektum"""
11.     def __init__(self, conn):
12.         threading.Thread.__init__(self)
13.         self.connexion = conn           # a kapcsolati socket referenciája
14.
15.     def run(self):
16.         while 1:
17.             message_recu = self.connexion.recv(1024)
18.             print "*" + message_recu + "*"
19.             if message_recu =='' or message_recu.upper() == "FIN":
20.                 break
21.         # A <fogadás> thread itt fejeződik be.
22.         # Kényszerítjük a <kibocsátás> thread lezárását :
23.         th_E._Thread__stop()
24.         print "Leállt a kliens. Kapcsolat megszakítva."
25.         self.connexion.close()
26.
27. class ThreadEmission(threading.Thread):
28.     """üzenetek kibocsátását kezelő thread objektum"""
29.     def __init__(self, conn):
30.         threading.Thread.__init__(self)
```

75 A « chat » : számítógépek (billentyűzet) közvetítésével zajló « beszélgetés »-t jelent.

```
31.         self.connexion = conn           # réf. du socket de connexion
32.
33.     def run(self):
34.         while 1:
35.             message_emis = raw_input()
36.             self.connexion.send(message_emis)
37.
38. # Főprogram – Kapcsolat létrehozása :
39. connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40. try:
41.     connexion.connect((host, port))
42. except socket.error:
43.     print "A kapcsolat meghiúsult."
44.     sys.exit()
45. print "A kapcsolat létrejött a serverrel."
46.
47. # Párbeszéd a server-rel : két thread-et indítunk az üzenetek
48. # fogadásának és indításának egymástól független kezelésére :
49. th_E = ThreadEmission(connexion)
50. th_R = ThreadReception(connexion)
51. th_E.start()
52. th_R.start()
```

Magyarázatok:

- **Általános megjegyzés** : Ebben a példában két – a fő-threadtől független - thread-objektum létrehozásáról döntöttem azért, hogy jól megvilágítsam a folyamatokat. A programunk összesen három threadet használ, amire a figyelmes olvasó megjegyzi, hogy kettő elég lenne. Valóban, a fő-threadet végül is csak a másik két thread elindítására használjuk ! Viszont semmilyen érdek nem szól a threadek számának korlátozása mellett. Ellenkezőleg : attól a pillanattól kezdve, ahogy ennek a technikának a használata mellett döntünk, hasznot kell húznunk az alkalmazás jól elkülönülő részekre tagolásából.
- 7. sor : A ***threading*** modul egy egész sor – a threadek kezeléséhez fontos - osztály definícióját tartalmazza. Itt csak a **Thread()** osztályt használjuk, de egy másikat (a **Lock()** osztályt) fogjuk a későbbiekben használni, amikor a különböző konkurens thread-ek szinkronizációjával kell foglalkoznunk.
- 9. - 25. sorok : A **Thread()** osztályból leszármaztatott osztályok lényegében egy **run()** metódust fognak tartalmazni. Ebben helyezzük el a programnak azt a részét, amit specifikusan a threadre bízunk. Gyakran egy végtelen hurokról lesz szó, mint itt. Ennek a metódusnak a tartalmát egy független scriptnek tekinthetjük, ami az alkalmazásunk többi komponensével párhuzamosan hajtódik végre. Amikor ez a kód végrehatódott, a thread bezáródik.
- 16. - 20. sorok : Ez a hurok kezeli az üzenetfogadást. Minden iterrációban a 17. sorban egy új üzenetre várva megszakad az utasítássorozat. Azonban a program többi része nem fagy le, a többi thread függetlenül folytatja munkáját.
- 19. sor : A kilépést a hurokból egy 'fin' (akár kis-, akár nagybetűvel lehet írni) üzenet, vagy egy üres string (ez a helyzet, ha a partner szünteti meg a kapcsolatot) fogadása váltja ki. Néhány « takarító » utasítást hajt végre, majd a thread befejeződik.
- 23. sor : Amikor az üzenetek fogadása befejeződik, azt akarjuk, hogy a program többi része is fejeződjön be. Tehát kényszerítenünk kell a másik thread-objektum (amit az üzenetek kibocsátására hoztunk létre) bezárását. Ezt a kényszerített bezárást a **_Thread__stop()**[76]

76 Hogy a puristák megbosássanak nekem, készséggel elismerem, ez a módszer nem igazán javasolt egy thread kényszerített leállítására. Azért engedtem meg magamnak ezt az eljárást, hogy ne nehezítsem meg túlságosan ennek a bevezető anyagnak a megértését. Az igényes olvasó a bibliográfiában (lásd a 8. oldalt) megemlített referencia művek segítségével elmélyedhet ebben a kérdésben.

metódussal érhetjük el.

- 27 - 36. sorok : Ez az osztály egy másik thread-objektumot definiál, ami ez alkalommal egy végtelen hurkot tartalmaz. Ez az objektum csak az előző bekezdésben leírt metódussal szüntethető meg. A hurok minden iterrációjában az utasítássorozat egy billentyűbemenetre várva a 35. sorban meg fog szakadni, de ez nem akadályozza meg a többi threadet a munkája végzésében.
- 38. - 45. sorok : Ezeket a sorokat az előző scriptekből változatlanul vettem át
- 47. - 52. sorok : A két thread « gyermek »-objektum létrehozása és indítása. Jegyezzük meg, hogy az indítás inkább a **start()** beépített metódussal javasolt, mint a **run()** metódus (amit nekünk kell majd definiálni) közvetlen hívásával. Tudjunk arról is, hogy a **start()**-ot csak egyszer hívhatjuk (ha már leállítottuk, akkor nem lehet újra indítani egy thread-objektumot).

18.6 Több klienskapcsolatot párhuzamosan kezelő server

A következő script egy olyan servert hoz létre, ami az előző oldalakon leírttal megegyező típusú kliensek kapcsolatait tudja kezelni.

Magát a servert nem használjuk kommunikációra : a kliensek azok, amik egymással kommunikálnak a server közvetítésével. A server egy kapcsoló szerepét játssza : elfogadja a kliensek rákapcsolódását, majd vár az üzeneteikre. Amikor egy bizonyos klienstől származó üzenet érkezik be, akkor a server hozzárak egy azonosító karakterláncot, ami jellemző a küldő kliensre, és kiküldi az összes többi kliensnek, hogy mindegyik láthasson minden üzenetet és tudhassa, hogy az kitől származik.

```
1.  # Egy egyszerűsített CHAT rendszert kezelő hálózati szerver definíciója.
2.  # thread-eket használ a klienskapcsolatok párhuzamos kezelésére.
3.
4.  HOST = '192.168.0.235'
5.  PORT = 40000
6.
7.  import socket, sys, threading
8.
9.  class ThreadClient(threading.Thread):
10.         '''thread-objektum leszármaztatása a klienssel való kapcsolat
    kezeléséhez'''
11.     def __init__(self, conn):
12.         threading.Thread.__init__(self)
13.         self.connexion = conn
14.
15.     def run(self):
16.         # Párbeszéd a klienssel :
17.         nom = self.getName()        # Minden thread-nek neve van
18.         while 1:
19.             msgClient = self.connexion.recv(1024)
20.             if msgClient.upper() == "FIN" or msgClient =="":
21.                 break
22.             message = "%s> %s" % (nom, msgClient)
23.             print message
24.             # Az üzenetet kiküldi az összes többi kliensnek :
25.             for cle in conn_client:
26.                 if cle != nom:      # ne küldje el a kibocsátónak
27.                     conn_client[cle].send(message)
28.
29.         # A kapcsolat zárása :
30.         self.connexion.close()      # a serveroldali kapcsolat megszakítása
31.         del conn_client[nom]        # törli a bejegyzését a szótárban
32.         print "Kliens %s lekapcsolódott." % nom
```

```
33.         # A thread itt fejeződik be
34.
35. # A server inicializálása – Socket létrehozása :
36. mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
37. try:
38.     mySocket.bind((HOST, PORT))
39. except socket.error:
40.     print "A socket összekapcsolása a választott címmel meghiúsult."
41.     sys.exit()
42. print "A server kész, várakozás a kérésekre ..."
43. mySocket.listen(5)
44.
45. # A kliensek által kért kapcsolatok várása és kezelése :
46. conn_client = {}                # a klienskapcsolatok szótára
47. while 1:
48.     connexion, adresse = mySocket.accept()
49.     # Új thread-objektum létrehozása a kapcsolat kezelésére   :
50.     th = ThreadClient(connexion)
51.     th.start()
52.     # A kapcsolat tárolása a szótárban :
53.     it = th.getName()        # thread-azonosító
54.     conn_client[it] = connexion
55.     print "Kliens %s felkapcsolódott, IP cím %s, port %s." %\
56.           (it, adresse[0], adresse[1])
57.     # Párbeszéd a klienssel :
58.     connexion.send("Ön felkapcsolódott. Küldje el az üzeneteit.")
```

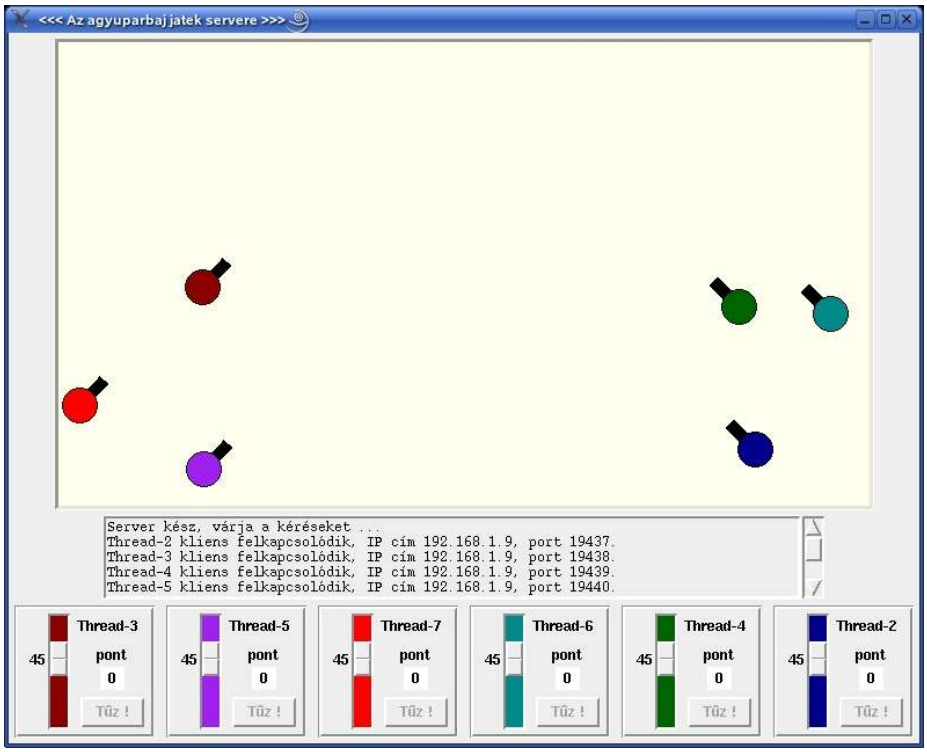
Magyarázatok :

- 35. - 43. sorok : A server inicializálása megegyezik a fejezet elején leírt alapserver inicializálásával.
- 46. sor : A különböző kapcsolatok hivatkozásait tárolni kell. Elhelyezhetnénk őket egy listában is, azonban ésszerűbb őket egy szótárba tenni. Ennek két oka van. Az első ok az, hogy tetszőleges sorrendben kell majd tudnunk hozzáadni vagy eltávolítani ezeket a hívatkozásokat, mert a kliensek tetszésük szerint kapcsolódnak föl és le. A második ok az, hogy mindegyik kapcsolatnál könnyen szert tehetünk egy egyedi azonosítóra, ami kulcs lehet egy szótárban. Ezt az azonosítót a **Thread()** osztály automatikusan fogja generálni.
- 47. - 51. sorok : A programban itt egy végtelen hurok van, ami állandóan új kapcsolatokra vár. Minden új kapcsolat számára létrehoz egy új **ThreadClient()** objektumot, ami az összes többitől függetlenül fog a kapcsolattal foglalkozni.
- 52. - 54. sorok : Egyedi azonosító előállítása a **getName()** metódussal. Itt azt a tényt használjuk ki, hogy a Python minden új thread-hez automatikusan hozzárendel egy egyedi nevet : ez a név mint azonosító (vagy kulcs) jó arra, hogy a szótárunkban megtaláljuk a megfelelő kapcsolatot. Megfigyelhető, hogy egy « Thread-N » formájú stringről van szó (ahol N a thread sorszáma).
- 15. - 17. sorok : Tartsuk észben, hogy annyi **ThreadClient()** objektum lesz létrehozva, ahány kapcsolat van és ezek az objektumok párhuzamosan fognak működni. A **getName()** metódust ezen objektumok bármelyikében felhasználhatjuk arra, hogy megkapjuk az objektum azonosítóját. Ezt az információt arra fogjuk használni, hogy az aktuális kapcsolatot megkülönböztessük a többitől (lásd a 26. sort).
- 18. - 23. sorok : A thread egy bizonyos klienstől származó összes üzenet fogadására szolgál. Ehhez egy végtelen ciklus kell, ami csak a « fin » speciális üzenet vagy egy üres üzenet (az az eset, amikor a kapcsolatot a partner szakítja meg) fogadására szakad meg.
- 24. - 27. sorok : Az egyik klienstől fogadott valamennyi üzenetet el kell küldeni az összes többi kliensnek. A kapcsolatok szótárának kulcskészletét egy **for** ciklussal járjuk be. A kulcsok fogják aztán lehetővé tenni, hogy megtaláljuk a kapcsolatokat. Egy egyszerű feltételvizsgálat (a 26.

sorban) megakadályozza, hogy visszaküldjük az üzenetet annak a kliensnek, akitől az származik.

- 31. sor : Egy socket lezárásakor a hivatkozását törlölni kell a szótárból, mert a hivatkozás már nem használható. Ezt minden különösebb óvintézkedés nélkül megtehetjük, mert a szótár elemei nem rendezettek (bármilyen sorrendben hozzáadhatunk és eltávolíthatunk elemeket).

18.7 Ágyúpárbaj – hálózati változat



A 15. fejezetben elmagyaráztam egy harci játék fejlesztését, amiben a játékosok ágyúkkal küzdöttek egymás ellen. A játék nem túl érdekes, amíg csak egyetlen számítógépen játszható. A tanult technikák beépítésével fogjuk tökéletesíteni. A teljes alkalmazás két programból fog állni, mint az előző oldalakon leírt « chat » alkalmazás : egy server alkalmazásból, amit csak egy gépen fogunk működtetni és egy kliens alkalmazásból, amit több más gépen elindíthatunk. A Python portábilis jellegéből adódóan különböző operációs rendszerekkel (MacOS <> Linux <> Windows !) működő számítógépek között rendezhetünk ágyúcsatákat.

18.7.1 Serverprogram: áttekintés

A server- és kliensprogramok ugyanazt az alapprogramot használják, amit nagyrészt a 15. fejezetben már kifejlesztett programból szedtem össze. A továbbiakban feltételezem, hogy a játék két előző verzióját az aktuális könyvtárban a **canon03.py** és **canon04.py** modulfile-okba mentettük. Az ***importálás*** és az ***öröklés*** okos felhasználásával a kód jó részét újra felhasználhatjuk.

A **canon04** modulból a **Canon()** osztály mind a server-, mind a kliensprogramnál ugyanolyan jól újra felhasználható. Ugyanebből a modulból fogjuk importálni az **AppAgyuParbaj()** osztályt is, amiből az alkalmazásserverünk master osztályát az **AppServer()**-t származtatjuk le. Konstatálni fogjuk, hogy ez utóbbiból állítjuk elő örökléssel az **AppClient()** alosztályt.

A **canon03** modulból fogjuk venni a **VezerloPult()** osztályt, amiből egy « távvezérlésre » alkalmasabb verziót készítünk.

Végül két új osztály jön az előzőekhez, melyek mindegyike egy *thread*-objektum létrehozására van specializálva : a **ThreadClients()** osztály, aminek egy példánya állandóan figyelni fogja az új kliensek kapcsolatkérésének fogadására fenntartott *socket*-et és a **ThreadConnexion()** osztály, ami arra szolgál, hogy annyi *socket*-objektumot hoz létre, amennyi a már felkapcsolódott kliensekkel való párbeszéd folytatásához szükséges.

Ezeket az új osztályokat a « chat » serverünkhöz - az előző oldalakon - kifejlesztett osztályok inspirálják. A fő különbség az előzőbbihez képest az, hogy egy speciális thread-et kell aktiválnunk a kliensre való várakozást és a klienskapcsolatokat kezelő kódnak, hogy a főalkalmazás ez idő alatt mást tudjon csinálni.

Innentől kezdve a feladatunknk abból áll, hogy a server és kliensei közötti dialógus számára ***kifejlesztünk egy kommunikációs protokolt***. Miről van szó ? Azoknak az üzeneteknek a tartalmát kell definiálni, amiket az összekapcsolt gépek egymással fognak váltani. Ez a « nyelv » lépésről lépésre fejleszthető. Egy alap párbeszéd megalkotásával kezdjük, majd apránként egészítjük ki bővebb « szókinccsel ».

A « chat » rendszer számára - az előző fejezetbenben - kifejlesztett kliensprogram felhasználásával megoldható a feladat lényegi része. Az említett kliensprogramot arra használjuk, hogy a fejlesztés alatt álló servernek « parancsokat » küldünk és addig javítjuk a servert, amíg megfelelően nem működik : vagyis, a serveren lépésről lépésre implementált eljárásoknak a kliens által « «manuálisan » kiküldött megfelelő üzenetekre adott válaszait teszteljük.

18.7.2 Kommunikációs protokol

Magától értetődik, hogy a későbbiekben leírt protokol teljesen önkényes. Más, teljesen eltérő konvenciókat is lehetne választani. A választásaink természetesen kritizálhatók és az olvasó talán más, hatékonyabb és egyszerűbb protokollal kívánja őket helyettesíteni.

Azt már tudjuk, hogy az üzenetek egyszerű karakterláncok. Előre látva, hogy egyes üzeneteknek egyszerre több információt kell továbbítani, úgy döntöttem, hogy mindegyik üzenet több, vesszőkkel elválasztott mezőt tartalmazhat. Bármelyik üzenet fogadásakor a mezőik tartalmát a beépített **split()** metódussal könnyen összegyűjthetjük egy listában.

A következőkben a kliens-server dialógusra látunk egy példát, ahogyan az a kliensoldalon követhető. A csillagok közötti üzeneteket a server küldi ; a többieket a kliens :

```
1.  *server OK*
2.  client OK
3.  *ágyúk,Thread-3;104;228;1;dark red,Thread-2;454;166;-1;dark blue,*
4.  OK
5.  *új_ágyú,Thread-4,481,245,-1,dark green,az_öné*
6.  célzás,25,
7.  tűz
8.  *elmozdul,Thread-4,549,280,*
9.  tűz
10. *elmozdul,Thread-4,504,278,*
11. *pontszám,Thread-4;1,Thread-3;-1,Thread-2;0,*
12. *szög,Thread-2,23,*
13. *szög,Thread-2,20,*
14. *tüzel,Thread-2,*
15. *elmozdul,Thread-2,407,191,*
16. *megszűnik,Thread-2*
17. *új_ágyú,Thread-5,502,276,-1,dark green*
```

Amikor elindul egy új kliens, akkor az egy kapcsolatkérést küld a servernek. Erre a server válaszul a « server OK » üzenetet küldi. Ennek nyugtázásául a kliens a « client OK » üzenetet küldi. Ez az üzenetváltás nélkülözhető, viszont lehetővé teszi annak ellenőrzését, hogy a kétirányú kommunikáció rendben megy-e. Miután a kliens figyelmezette a servert, hogy készen áll a párbeszédre, a server elküldi neki a játékban már jelen lévő (esetleg egy sincs jelen) ágyúk leírását : azonosító, pozíció a vásznon, írány és szín (3. sor).

A kliens vételi nyugtájára válaszul (4. sor) a server egy új ágyút helyez el a játéktéren, majd az összes felkapcsolódott klienst, - nemcsak azt, amelyik előidézte az új ágyú telepítését - értesíti a telepítés jellemzőiről. Az új kliensnek küldött üzenetben azonban van egy eltérés (mivel ő a tulajdonosa az új ágyúnak) : az ágyú jellemzőin túl, amik mindenkinek meg vannak adva, tartalmaz egy kiegészítő mezőt, ami az « az_öné » szöveget tartalmazza (hasonlítsuk össze például az 5. sort a 17. sorral, amelyik egy másik játékost értesít a kapcsolatról.) Ez a kiegészítő jelzés teszi lehetővé az ágyútulajdonos kliensnek, hogy a hasonló üzenetek között meg tudja különböztetni azt, amelyik a server által neki adott egyedi azonosítót tartalmazza.

A 6. és 7. sor üzenetei a kliens által küldött parancsok (dőlésszög beállítása és tűzparancs). A játék előző verziójában már megállapodtunk abban, hogy a lövés után az ágyúk (véletlenszerűen) egy kicsit elmozdulnak. A server tehát végrehajtja ezt a műveletet és utána minden felkapcsolódott klienssel ismerteti az eredményt. A 8. sorban a servertől kapott üzenet tehát egy ilyen elmozdulásnak a jelzése (a megadott koordináták az illető ágyú koordinátái).

A 11. sor azt a serverüzenetet mutatja be, amikor eltaláltak egy célt. Minden játékos új pontszámát is közli minden klienssel.

A 12., 13. és 14. sorok serverüzenetei egy másik játékos által elindított akciót jeleznek (dőlésszög beállítása, amit lövés követ). A lövés leadása után az illető ágyú véletlenszerűen elmozdul (15. sor).

16. és 17. sor : amikor az egyik kliens megszakítja a kapcsolatot, a server erről értesíti a többi klienst azért, hogy a megfelelő ágyú mindegyik poszton eltűnjön a játéktérről. Megfordítva: új kliensek bármelyik pillanatban felkapcsolódhatnak, hogy részt vegyenek a játékban.

Kiegészítő megjegyzések :

Minden üzenet első mezője megjelöli az üzenet tartalmát. A kliens által küldött üzenetek nagyon egyszerűek : a játékos különböző akcióinak (a lövés szögének módosítása és a tűzparancs) felelnek meg. A server üzenetei egy kicsit összetettebbek. A server a többségüket minden felkapcsolódott kliensnek elküldi, hogy tájékoztassa őket a játék lefolyásáról. Következésként ezek az üzenetek meg kell hogy említsék annak a játákosnak az azonosítóját, aki parancsot adott egy akcióra vagy aki

valamilyen változásban érintett. Föntebb láttuk, hogy ezek az azonosítók azok a nevek, amiket a server thread-kezelője – minden alkalommal, amikor új kliens kapcsolódik a serverhez – automatikusan generál.

Bizonyos, a játék egészére vonatkozó utasítások mezőnként több információt tartalmaznak. Ebben az esetben a különböző « almezőket » pontosvessző választja el egymástól (3. és 11. sor).

18.7.3 Serverprogram : első rész

A következő oldalakon található a komplett serverprogram. Három egymást követő részletben mutatom be, hogy a megfelelő kódrészlethez közelebb kerüljön a magyarázat. A sorok számozása folyamatos. Jóllehet a program már viszonylag hosszú és összetett, az olvasó valószínűleg úgy fogja gondolni, hogy még tökéletesítésre szorul az általános bemutatás szintjén. Az olvasóra hagyom, tegye hozzá azokat a kiegészítéseket, amik hasznosnak tűnnek (például indításkor felkínálni a host gép adatainak kiválasztását, egy menüsort, stb.) :

```
1.  ##########################################################
2.  # Agyuparbaj - server resz                               #
3.  # (C) Gerard Swinnen, Verviers (Belgique) - July 2004 #
4.  # Licence : GPL                                          #
5.  # A script vegrehajtasa elott ellenorizze, hogy az    #
6.  # alabbi IP cim a host IP cime.                          #
7.  # Valaszthat egy eltero portszamot, vagy megvaltoztat-#
8.  # hatja a jatekter mereteit.                             #
9.  # Minden esetben ellenorizze, hogy ugyanezeket a val- #
10. # toztatasokat elvegezte-e minden  kliens scripten    #
11. ##########################################################
12.
13. host, port = '192.168.1.9', 35000
14. width_, height_ = 700, 400             # a jatekter meretei
15.
16. from Tkinter import *
17. import socket, sys, threading, time
18. import canon03
19. from canon04 import Canon, AppAgyuParbaj
20.
21. class VezerloPult(canon03.VezerloPult):
22.     """Tokeletesitett vezerlo pult"""
23.     def __init__(self, boss, canon):
24.         canon03.VezerloPult.__init__(self, boss, canon)
25.
26.     def tuzel(self):
27.         "az asszocialt agyu lovesenek elinditasa"
28.         self.appli.tuzel_agyu(self.canon.id)
29.
30.     def iranyzas(self, angle):
31.         "az asszocialt agyu dolesszogenek beallitasa"
32.         self.appli.iranyzas_agyu(self.canon.id, angle)
33.
34.     def pont_ertek(self, sc =None):
35.         "új <sc> pontszam kiirasa, vagy a letezo olvasasa"
36.         if sc == None:
37.             return self.score
38.         else:
39.             self.score =sc
40.             self.pontok.config(text = ' %s ' % self.score)
41.
42.     def inactivate(self):
43.         "a tuzgomb es a szogbeallito rendszer inaktivalasa"
44.         self.bTuz.config(state =DISABLED)
45.         self.celzas.config(state =DISABLED)
46.
47.     def activate(self):
48.         "a tuzgomb es a szogbeallito rendszer aktivalasa"
```

```
49.         self.bTuz.config(state =NORMAL)
50.         self.celzas.config(state =NORMAL)
51.
52.     def beallitas(self, angle):
53.         "a kurzor poziciojanak megvaltoztatasa"
54.         self.celzas.config(state =NORMAL)
55.         self.celzas.set(angle)
56.         self.celzas.config(state =DISABLED)
57.
```

A **VezerloPult()** osztályt a **canon03** modulból importált azonos nevű osztályból leszármaztatással hozzuk létre. A szülőosztály minden jellemzőjét örökli, de ***felül kell írnunk***[77] a **tuzel()** és az **iranyzas ()** metódusokat :

A program egygépes változatában mindegyik vezérlőpult közvetlenül tudta irányítani a megfelelő ágyút. Ebben a hálózati változatban viszont a kliensek azok, akik távolról vezérlik az ágyúk működését. Következésként a server ablakában megjelenő vezérlőpultok csak másolhatják a játékosok által az egyes klienseken végrehajtott manővereket. A tűzgomb és a lövésszög beállító cursor tehát inaktiválva vannak, de a kijelzések a főalkalmazás által nekik címzett parancsoknak vannak alávetve.

Ezt az új **VezerloPult()** osztályt egyformán fogjuk használni a kliensprogram mindegyik példányában úgy, ahogy van. A kliens ablakában is, mint a server ablakában minden vezérlőpult másolat lesz– azonban a játékos ágyújának vezérlőpultja teljesen működőképes lesz.

Ezek az okok is magyarázzák az : **activate()**, **inactivate()**, **beallitas()** és **pont_ertek()** új metódusok megjelenését is. A főalkalmazás ezeket is hívni fogja a server és a kliensei közötti üzenet-utasítás cserére válaszul.

A **ThreadConnexion()** osztály thread-objektumok sorozatának generálására való, amik párhuzamosan fognak foglalkozni a kliensek által létesített összes kapcsolattal. Az osztály **run()** metódusa tartalmazza a server központi funkcióját, tudni illik azt az utasításhurkot, ami egy specifikus klienstől származó utasítások fogadását kezeli, mely utasítások mindegyike reakciók sorozatát vonja maga után. Itt fogjuk megtalálni az előző oldalakon leírt kommunikációs protokol konkrét felhasználását (bizonyos utasításokat viszont a későbbiekben tárgyalt **AppServer()** osztály **agyu_veletlen_elmozditasa()** és **goal()** metódusai generálnak).

```
58. class ThreadConnexion(threading.Thread):
59.     """egy klienskapcsolatot kezelo thread objektum"""
60.     def __init__(self, boss, conn):
61.         threading.Thread.__init__(self)
62.         self.connection = conn        # a kapcsolat socket-jenek hivatkozasa
63.         self.app = boss                 # az alkalmazasablak hivatkozasa
64.
65.     def run(self):
66.         "a klienstol kapott uzenetre valaszul vegrehajtott akciok"
67.         name = self.getName()            # kliens id-je = a thread neve
68.         while 1:
69.             msgClient = self.connection.recv(1024)
70.             print "**%s** %s" % (msgClient, name)
71.             deb = msgClient.split(',')[0]
72.             if deb == "fin" or deb =="":
73.                 self.app.agyu_eltavolitas(name)
74.                 # a tobbi kliens ertesitese az agyu eltavolitasarol :
75.                 self.app.locking.acquire()
76.                 for cli in self.app.conn_client:
77.                     if cli != name:
78.                         message = "megszűnik,%s" % name
```

77 Ismétlés : egy szülőosztálybeli metódus nevén a leszármaztatott osztályban új metódust definiálhatunk azért, hogy módosítsuk a metódus működését a leszármaztatott osztályban. Ezt **felülírás**nak nevezzük (lásd 169 . oldalt).

```
79.                     self.app.conn_client[cli].send(message)
80.                 self.app.locking.release()
81.                 # az aktualis thread zarasa :
82.                 break
83.             elif deb =="client OK":
84.                 # kozli az uj klienssel a mar bejegyzett agyukat :
85.                 msg ="ágyúk,"
86.                 for g in self.app.guns:
87.                     gun = self.app.guns[g]
88.                     msg =msg +"%s;%s;%s;%s;%s," % \
89.                             (gun.id, gun.x1, gun.y1, gun.irany, gun.szin)
90.                 self.app.locking.acquire()
91.                 self.connection.send(msg)
92.                 # fogadas nyugtazasat varja ('OK') :
93.                 self.connection.recv(100)
94.                 self.app.locking.release()
95.                 # a server jatekterehez hozzaad egy agyut.
96.                 # az agyu jellemzoit adja meg visszateresi ertekkent :
97.                 x, y, irany, szin = self.app.agyu_hozzadasa(name)
98.                 # ennek az uj agyunak a jellemzoit megadja az osszes tobbi
99.                 # mar felkapcsolodott kliensnek :
100.                self.app.locking.acquire()
101.                for cli in self.app.conn_client:
102.                    msg ="új_ágyú,%s,%s,%s,%s,%s" % \
103.                                    (name, x, y, irany, szin)
104.                    # az uj kliensnek egy mezot ad, ami jelzi, hogy
105.                    # az uzenet az o agyujara vonatkozik :
106.                    if cli == name:
107.                        msg =msg +",az_öné"
108.                    self.app.conn_client[cli].send(msg)
109.                self.app.locking.release()
110.            elif deb =='tűz':
111.                self.app.tuzel_agyu(name)
112.                # jelzi ezt a lovest az osszes tobbi kliensnek :
113.                self.app.locking.acquire()
114.                for cli in self.app.conn_client:
115.                    if cli != name:
116.                        message = "tüzel,%s," % name
117.                        self.app.conn_client[cli].send(message)
118.                self.app.locking.release()
119.            elif deb =="célzás":
120.                t =msgClient.split(',')
121.                # tobb szoget kaphatunk, hasznaljuk az utolsot :
122.                self.app.iranyzas_agyu(name, t[-2])
123.                # Az osszes tobbi kliensnek jelezzuk a valtozast :
124.                self.app.locking.acquire()
125.                for cli in self.app.conn_client:
126.                    if cli != name:
127.                        # záró vessző, az uzenetek neha csop. vannak :
128.                        message = "szög,%s,%s," % (name, t[-2])
129.                        self.app.conn_client[cli].send(message)
130.                self.app.locking.release()
131.
132.        # Kapcsolat zarasa :
133.        self.connection.close()          # megszakitja a kapcsolatot
134.        del self.app.conn_client[name] # a szotarban toroljuk a hivatk.-t
135.        self.app.kiir("A %s kliens lekapcsolódik.\n" % name)
136.        # A thread itt fejezodik be
```

18.7.4 Konkurens thread-ek szinkronizálása « zárolással » (thread locks)

A fenti kód vizsgálata során az olvasó bizonyára észrevette azoknak az utasításblokkoknak a speciális struktúráját, amikkel a server ugyanazt az üzenetet küldi az összes kliensének. Nézzük meg például a 74. - 80. sorokat :

A 75. sor a főalkalmazás konstruktora által létrehozott « zár » objektum **acquire()** metódusát aktiválja. (Lásd később). Ez az objektum a script elején importált **Lock()** osztály – mely a threading modulnak része - egy példánya. A következő (76 - 79) sorok egy üzenet küldését idézik elő (egy kivételével) az összes kapcsolódott kliensnek. Utána a « zár » objektum **release()** metódusa lesz aktiválva.

Mire való ez a « zár » objektum ? Mivel a threading modul egyik osztálya hozza létre, kitalálható, hogy a thread-ekkel kapcsolatos a használata. Valóban, az ilyen « zár » objektumok a ***konkurens thread-ek szinkronizálására*** valók. Miről van szó ?

Tudjuk, hogy a server mindegyik hozzákapcsolódó kliensnek egy másik thread-et indít el. Utána ezek a thread-ek párhuzamosan működnek. Fennáll tehát annak a veszélye, hogy időnként kettő vagy több thread egyidejűleg próbál meg egy közös erőforrást használni.

Azokban a kódsorokban, amiket például megbeszéltünk, egy olyan thread-del van dolgunk, ami kvázi az összes kapcsolatot egy üzenet küldésére akarja felhasználni. Lehetséges, hogy ez alatt az idő alatt egy másik thread is megkísérli egyik vagy másik kapcsolatot használni, ami azzal a veszéllyel jár, hogy működési zavar léphet fel (esetünkben több üzenet kaotikus szuperpozícióját okozhatja).

A ***thread-ek közötti konkurálás*** problémáját egy « zár » objektum (*thread lock objektum*) alkalmazásával lehet megoldani. Az összes konkurens thread által hozzáférhető névtérben csak egyetlen ilyen objektumot hozunk létre. Jellemzője, hogy mindíg vagy a ***locked,*** vagy ***unlocked*** állapotok valamelyikében található. Kiindulási állapota ***unlocked.***

Használata :

Amikor egy thread egy közös erőforrás használatára készül, először a zár **acquire()** metódusát aktiválja. Ha az unlock-olva volt, akkor lock-ol és a kérő thread nyugodtan használhatja a közös erőforrást. Amikor befejezte az erőforrás használatát, akkor aktiválja a zár **release()** metódusát, ami a zárat átviszi az unlocked állapotba.

Ha egy másik konkurens thread is aktiválja a zár **acquire()** metódusát, amikor a zár locked állapotban van, a metódus « nem adja a kezét », blokkolva ez utóbbi thread-et, ami felfüggeszti a működését mindaddig, amíg a retesz át nem megy unlocked állapotba. Ez tehát megakadályozza a közös erőforráshoz való hozzáférést amíg azt egy másik thread használja. Amikor a retesz unlock-olva van az egyik várakozó thread (több is lehet belőlük) folytatja működését és így tovább.

A « zár » objektum úgy tárolja a blokkolt thread-ek hívatkozásait, hogy a **release()** metódusa hívásakor csak egy blokkolt thread-et szabadít fel. Mindíg ügyelni kell rá, hogy mindegyik thread, amelyik egy erőforráshoz történő hozzáférés előtt a retesz **acquire()** metódusát aktíválja, utána a **release()** metódust is aktiválja.

Amennyiben mindegyik konkurens thread betartja ezt az eljárást, úgy ez az egyszerű technika megakadályozza azt, hogy egy közös erőforrást egyidejűleg több thread használjon. Ebben az esetben azt mondjuk, hogy a thread-ek ***szinkronizálva vannak.***

18.7.5 Serverprogram : folytatás és befejezés

Az alábbi két osztály teljessé teszi a serverscript-et. A **ThreadClients()** osztályban implementált kód meglehetősen hasonlít a « Chat »program programtörzse számára fejlesztett kódunkhoz. Jelen esetben azonban a kódot egy **Thread()** osztályból leszármaztatott osztályban fogjuk elhelyezni, mert ezt a kódot a főalkalmazás thread-jétől független thread-ben kell működtetnünk. Ezt a thread-et kizárólag a grafikus interface **mainloop()** ciklusa használja[78].

Az **AppServer()** osztály a **canon04** modul **AppAgyuParbaj()** osztályából származik. Kiegészítő metódusokat adtam hozzá, amiket alkalmassá tettem a kliensekkel folytatott párbeszédekből eredő minden műveletet végrehajtására. Föntebb már említettem, hogy a kliensek mindegyike létre fog hozni egy ebből az osztályból leszármaztatott változatot (azért, hogy ugyanazokat az ablak, vászon, stb. definíciókat használjuk).

```
138.class ThreadClients(threading.Thread):
139.    """az uj klienskapcsolatokat kezelo thread-objektum"""
140.    def __init__(self, boss, connec):
141.        threading.Thread.__init__(self)
142.        self.boss = boss                # az alkalmazasablak hivatkozasa
143.        self.connec = connec            # a kezdo socket hivatkozasa
144.
145.    def run(self):
146.        "uj klienskapcsolatok varasa es kezelese "
147.        txt ="Server kész, várja a kéréseket ...\n"
148.        self.boss.kiir(txt)
149.        self.connec.listen(5)
150.        # A kliensek altal kert kapcsolatok kezelese :
151.        while 1:
152.            new_conn, adresse = self.connec.accept()
153.            # Egy uj thread-objektum letrehozasa a kapcsolat kezelesere :
154.            th = ThreadConnexion(self.boss, new_conn)
155.            th.start()
156.            it = th.getName()        # a thread egyedi azonositoja
157.            # A kapcsolat tarolasa a szotarban :
158.            self.boss.kapcsolat_bejegyzese(new_conn, it)
159.            # Kiir :
160.            txt = "%s kliens felkapcsolódik, IP cím %s, port %s.\n" %\
161.                   (it, adresse[0], adresse[1])
162.            self.boss.kiir(txt)
163.            # A parbeszed elkezdese a klienssel :
164.            new_conn.send("server OK")
165.
166.class AppServer(AppAgyuParbaj):
167.    """az alkalmazas foablaka (server vagy kliens)"""
168.    def __init__(self, host, port, width_c, height_c):
169.        self.host, self.port = host, port
170.        AppAgyuParbaj.__init__(self, width_c, height_c)
171.        self.active =1          # aktivitas-flag
172.        # ugyeljunk a megfelelo kilepesre ha bezarjuk az ablakot :
173.        self.bind('<Destroy>',self.threadek_zarasa)
174.
175.    def specificites(self):
176.        "a server resz specialis objektumainak elkeszitese"
177.        self.master.title('<<< Az agyuparbaj jatek servere >>>')
178.
179.        # egy gorgeto savval asszocialt Text widget :
180.        st =Frame(self)
181.        self.notice =Text(st, width =85, height =5)
182.        self.notice.pack(side =LEFT)
183.        scroll =Scrollbar(st, command =self.notice.yview)
184.        self.notice.configure(yscrollcommand =scroll.set)
```

78 Ezt a kérdést néhány oldallal később fogom részletezni, mert érdekes perspektívákat nyit meg.
Lásd : Animációk optimalizálása thread-ek segítségével, 303. oldal.

```
185.         scroll.pack(side =RIGHT, fill =Y)
186.         st.pack()
187.
188.         # halozati server resz :
189.         self.conn_client = {}              # klienskapcsolatok szotara
190.         self.locking =threading.Lock()     #zár a thread-ek szinkroniz.-hoz
191.         # A server inicializalasa - Socket letrehozasa:
192.         connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
193.         try:
194.             connection.bind((self.host, self.port))
195.         except socket.error:
196.             txt ="Socket kapcsol. a %s host, %s porthoz meghiúsult.\n" %\
197.                   (self.host, self.port)
198.             self.notice.insert(END, txt)
199.             self.reception =None
200.         else:
201.             # a kliensek kapcsolodasat figyelo thread inditasa :
202.             self.reception = ThreadClients(self, connection)
203.             self.reception.start()
204.
205.     def agyu_veletlen_mozditasa(self, id):
206.         "az <id> agyu veletlenszeru elmozditasa"
207.         x, y = AppAgyuParbaj.agyu_veletlen_mozditasa(self, id)
208.         # az uj koordinatak kozlese az osszes klienssel :
209.         self.locking.acquire()
210.         for cli in self.conn_client:
211.             message = "elmozdul,%s,%s,%s," % (id, x, y)
212.             self.conn_client[cli].send(message)
213.         self.locking.release()
214.
215.     def goal(self, i, j):
216.         "az <i> agyu jelzi, hogy eltalalta a <j> ellenfelet"
217.         AppAgyuParbaj.goal(self, i, j)
218.         # az osszes klienssel tudatjuk az uj pontszamokat :
219.         self.locking.acquire()
220.         for cli in self.conn_client:
221.             msg ='pontszám,'
222.             for id in self.pult:
223.                 sc = self.pult[id].pont_ertek()
224.                 msg = msg +"%s;%s," % (id, sc)
225.             self.conn_client[cli].send(msg)
226.         time.sleep(.5)           # az üzenetek jobb szeparalasa erdekeben
227.         self.locking.release()
228.
229.     def agyu_hozzadasa(self, id):
230.         "egy <id> nevu agyu es vezerlopult letrehozasa 2 szotarban"
231.         # valtogatni fogjuk a 2 taboret :
232.         n = len(self.guns)
233.         if n %2 ==0:
234.             irany = -1
235.         else:
236.             irany = 1
237.         x, y = self.veletlen_koord(irany)
238.         szin =('dark blue', 'dark red', 'dark green', 'purple',
239.                'dark cyan', 'red', 'cyan', 'orange', 'blue', 'violet')[n]
240.         self.guns[id] = Canon(self.jatek, id, x, y, irany, szin)
241.         self.pult[id] = VezerloPult(self, self.guns[id])
242.         self.pult[id].inactivate()
243.         return (x, y, irany, szin)
244.
245.     def agyu_eltavolitas(self, id):
246.         "az <id> agyu es vezerlopult eltavolitasa"
247.         if self.active == 0:        # az ablakot bezartuk
248.             return
249.         self.guns[id].torol()
250.         del self.guns[id]
251.         self.pult[id].destroy()
252.         del self.pult[id]
253.
254.     def iranyzas_agyu(self, id, angle):
```

```
255.         "az <id> agyu dolesszogenek <angle> ertekre valo beallitasa"
256.         self.guns[id].iranyzas(angle)
257.         self.pult[id].beallitas(angle)
258.
259.     def tuzel_agyu(self, id):
260.         "az <id> agyu lovesenek inditasa"
261.         self.guns[id].tuz()
262.
263.     def kapcsolat_bejegyzese(self, conn, it):
264.         "A kapcsolat tarolasa egy szotarba"
265.         self.conn_client[it] = conn
266.
267.     def kiir(self, txt):
268.         "egy üzenet kiirasa a szovegzonaba"
269.         self.notice.insert(END, txt)
270.
271.     def threadek_zarasa(self, evt):
272.         "a letezo kapcsolatok megszakitasa es a thread-ek zarasa"
273.         # az osszes klienssel letrehozott kapcsolat megszakitasa :
274.         for id in self.conn_client:
275.             self.conn_client[id].send('fin')
276.         # a keresekre varo server thread befejezesenek kenyszeritese :
277.         if self.reception != None:
278.             self.reception._Thread__stop()
279.         self.active =0        # megakad. a tovabbi hozzafereseket a Tk -hoz
280.
281. if __name__ =='__main__':
282.     AppServer(host, port, width_, height_).mainloop()
```

Magyarázatok :

- 173. sor : Alkalmanként megtörténik, hogy meg akarjuk változtatni az alkalmazás bezárásának a menetét - amit a felhasználó a programunkból való kilépéssel indít el - például azért, mert fontos adatok file-ba mentését, vagy más ablakok bezárását, stb. is ki akarjuk kényszeríteni. Ehhez elegendő a <Destroy> eseményt detektálni, ahogyan itt is eljárunk, hogy az összes aktív thread befejezését kikényszerítsük.
- 179. - 186. sorok : Íme, hogyan asszociálhatunk egy gördítő sávot (**Scrollbar** *widget*-et) egy **Text** *widget*-hez (ugyanezt megtehetjük egy **Canvas** *widget*-tel) a Pmw könyvtár hívása nélkül[79].
- 190. sor : A thread-ek szinkronizálását lehetővé tevő « zár »-objektum létrehozása.
- 202, 203. sorok : A potenciális kliensek kapcsolatkérését állandóan figyelő thread-objektum létrehozása.
- 205. - 213., 215. - 227. sorok : Ezek a metódusok a szülősztálytól örökölt azonos nevű metódusokat overload-olják. Először a szülősztály metódusát hívják, hogy azok elvégezzék a feladatukat (207. , 217. sorok), majd hozzáteszik a saját funkciójukat, ami abból áll, hogy mindenkinek jelzik, hogy mi történt.
- 229. - 243. sorok : Ez a metódus minden alkalommal, amikor egy új kliens kapcsolódik a serverhez, egy új lőállást hoz létre. Az ágyúk véletlenszerűen vannak elhelyezve felváltva a bal- és a jobboldalon. Az eljáráson természetesen lehetne javítani. Az előre megadott színlista 10-re korlátozza a kliensek számát, aminek elégnek kellene lenni.

79 Lásd : ***Python Mega Widget-ek***, 205. oldal .

18.7.6 Kliensprogram

Mint a serverprogram, ez is viszonylag rövid, mert modulokat importál és az osztályok öröklési mechanizmusát alkalmazza. A serverscript-et egy **canon_serveur.py** nevű filemodulba kellett menteni. Ezt a file-t az aktuális könyvtárba kell elhelyezni ugyanúgy, mint a **canon03.py** és **canon04.py** filemodulokat, amiket használ.

Az így importált modulokból a script a **Canon()** és a **VezerloPult()** osztályokat változatlan formában használja és egy **AppServer()** osztályból leszármaztatott osztályt is használ. Ez utóbbiban számos metódust felülírtunk, hogy adaptáljuk a működésüket. Tekintsük például a **goal()** és az **agyu_veletlen_elmozditasa()** metódusokat, amiknek felülírt változata semmit sem csinál (**pass** utasítás), mert a lövések után a találatok számlálását és az ágyúk elmozdítását csak a server végezheti.

A **ThreadSocket()** osztály **run()** metódusában (86 - 126. sorok) található a serverrel váltott üzeneteket kezelő kód. Egyébként benne hagytam egy **print** utasítást (a 88. sorban), hogy a server által fogadott utasítások megjelenjenek a standard outputon. Az olvasó természetesen törölheti ezt az utasítást, ha a játék végleges formáját készíti el.

```
1.  ######################################################################
2.  # Agyuparbaj - kliensresz                                            #
3.  # (C) Gerard Swinnen, Liege (Belgique) - Juillet 2004                 #
4.  # Licence : GPL                                                      #
5.  # A script vegrehajtasa elott ellenorizze, az alabbi IP cim a host   #
6.  # IP cime e. Valaszthat egy eltero portszamot, vagy megvaltoztathat- #
7.  # ja a jatekter mereteit.-Minden esetben ellenorizze, hogy ugyan     #
8.  # ezeket a valtoztatasokat elvegezte-e a kliens scripteken-          #
9.  ######################################################################
10.
11. from Tkinter import *
12. import socket, sys, threading, time
13. from szerver import Canon, VezerloPult, AppServer
14.
15. host, port = '192.168.1.9', 35000
16. width_, height_ = 700, 400          # jatekter meretei
17.
18. class AppClient(AppServer):
19.     def __init__(self, host, port, larg_c, haut_c):
20.         AppServer.__init__(self, host, port, larg_c, haut_c)
21.
22.     def specificites(self):
23.         "a kliens resz specifikus objektumainak elokeszitese"
24.         self.master.title('<<< Ágyúpárbaj >>>')
25.         self.connec =ThreadSocket(self, self.host, self.port)
26.         self.connec.start()
27.         self.id =None
28.
29.     def agyu_hozzadasa(self, id, x, y, irany, szin):
30.         "<id> nevu agyu es vezerlopult peldanyok letrehozasa 2 szotarban"
31.         self.guns[id] = Canon(self.jatek,id,int(x),int(y),int(irany),szin)
32.         self.pult[id] = VezerloPult(self, self.guns[id])
33.         self.pult[id].inactivate()
34.
35.     def activate_pupitre_personnel(self, id):
36.         self.id =id                          # a szervertol kapott azonosito
37.         self.pult[id].activate()
38.
39.     def tuzel_agyu(self, id):
40.         r = self.guns[id].tuz()      # False-ot ad vissza, ha blokkolva van
41.         if r and id == self.id:
42.             self.connec.report_gunfire()
43.
44.     def imposer_score(self, id, sc):
```

```
45.         self.pult[id].pont_ertek(int(sc))
46.
47.     def elmozdit_agyu(self, id, x, y):
48.         "megjegyzes: az x es y ertekeket stringkent fogadja"
49.         self.guns[id].elmozdit(int(x), int(y))
50.
51.     def iranyzas_agyu(self, id, angle):
52.         "az <id> agyu dolesszoget az <angle> ertekkel valtoztatja"
53.         self.guns[id].iranyzas(angle)
54.         if id == self.id:
55.             self.connec.report_angle(angle)
56.         else:
57.             self.pult[id].beallitas(angle)
58.
59.     def threadek_zarasa(self, evt):
60.         "a kapcsolatok megszuntetese es a threadek zarasa"
61.         self.connec.finish()
62.         self.active =0        # kesobbi hozzaferes megakadalyozasa a Tk -hoz
63.
64.     def agyu_veletlen_mozditasa(self, id):
65.         pass                              # => hatastalan metodus
66.
67.     def goal(self, a, b):
68.         pass                              # => hatastalan metodus
69.
70.
71. class ThreadSocket(threading.Thread):
72.     """a serverrel tort. uzenetvaltast kezelo thread-objektum letrehozasa"""
73.     def __init__(self, boss, host, port):
74.         threading.Thread.__init__(self)
75.         self.app = boss            # az alkalmazasablak hivatkozasa
76.         # socket letrehozasa - kapcsolodas a serverhez :
77.         self.connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
78.         try:
79.             self.connection.connect((host, port))
80.         except socket.error:
81.             print "A kapcsolat nem jött letre."
82.             sys.exit()
83.         print "Letrejott a kapcsolat a serverrel."
84.
85.     def run(self):
86.         while 1:
87.             msg_rece = self.connection.recv(1024)
88.             print "*%s*" % msg_rece
89.             # az uzenetet eloszor egy listava alakitja at :
90.             t =msg_rece.split(',')
91.             if t[0] =="" or t[0] =="fin":
92.                 # az aktualis thread zarasa:
93.                 break
94.             elif t[0] =="server OK":
95.                 self.connection.send("client OK")
96.             elif t[0] =="ágyúk":
97.                 self.connection.send("OK")       # accuse de reception
98.                 # a lista 1. es utolso elemet eliminaljuk.
99.                 # amik megmaradnak azok is listak :
100.                lc = t[1:-1]
101.                # mindegyik egy agyu komplett leirasa :
102.                for g in lc:
103.                    s = g.split(';')
104.                    self.app.agyu_hozzadasa(s[0], s[1], s[2], s[3], s[4])
105.            elif t[0] =="új_ágyú":
106.                self.app.agyu_hozzadasa(t[1], t[2], t[3], t[4], t[5])
107.                if len(t) >6:
108.                    self.app.activate_pupitre_personnel(t[1])
109.            elif t[0] =='angle':
110.                # lehet, hogy tobb, csoportba foglalt informaciot fogadott
111.                # ekkor csak az elsot tekintjuk :
112.                self.app.iranyzas_agyu(t[1], t[2])
113.            elif t[0] =="tüzel":
114.                self.app.tuzel_agyu(t[1])
```

```
115.             elif t[0] =="scores":
116.                 # a lista 1. es utolso elemet eliminaljuk.
117.                 # amik megmaradnak azok is listak :
118.                 lc = t[1:-1]
119.                 # mindegyik elem egy pontszám leírása :
120.                 for g in lc:
121.                     s = g.split(';')
122.                     self.app.imposer_score(s[0], s[1])
123.             elif t[0] =="elmozdul":
124.                 self.app.elmozdit_agyu(t[1],t[2],t[3])
125.             elif t[0] =="megszűnik":
126.                 self.app.agyu_eltavolitas(t[1])
127.
128.         # Itt fejezodik be a <reception> thread.
129.         print "Kliens leáll. Kapcsolatot megszakitja."
130.         self.connection.close()
131.
132.     def report_gunfire(self):
133.         self.connection.send('tűz')
134.
135.     def report_angle(self, angle):
136.         self.connection.send('célzás,%s,' % angle)
137.
138.     def finish(self):
139.         self.connection.send('fin')
140.
141. # Foprogramm :
142. if __name__ =='__main__':
143.     AppClient(host, port, width_, height_).mainloop()
144.
```

Magyarázatok :

- 15., 16. sor : Egy form hozzáadásával - ami ezeket az értékeket az indításkor a felhasználótól fogja kérni - az olvasó maga is tökéletesítheti a scriptet.
- 19. - 27. sorok : A szülőosztály constructora a **specificites()** metódus hívásával fejeződik be. Ez utóbbiban elhelyezhetjük mindazt, amit máshogy kell a serverben illetve a kliensekben megszerkeszteni. (Nevezetesen : a server létrehoz egy **text** *widget*-et, amit a kliensekben nem találunk meg ; mindkettő különböző thread-objektumokat indít a kapcsolatok kezelésére.
- 39 - 42. sorok : Amikor a felhasználó megnyomja a tűzgombot, minden esetben ennek a metódusnak a hívására kerül sor. Azonban az ágyú nem adhat le sorozatlövéseket. Következésként új lövés leadására nem kerülhet addig sor, amíg az előző lövedék nem fejezte be a röppályáját. Az ágyú-objektum tuz() metódusának « igaz » vagy « hamis » visszatérési értéke az ami jelzi, hogy a lövés el lett fogadva vagy sem. Ezt az értéket csak arra használjuk, hogy a servernek (és a klienseknek) jelezzük, hogy valóban lövések voltak.
- 105. - 108. sorok : Minden alkalommal, amikor új kliens kapcsolódik fel, mindegyik játéktérhez egy új ágyút kell hozzáadni (vagyis a server vásznához és minden felkapcsolódott kliens vásznához). Ebben a pillanatban a server ugyanazt az üzenetet küldi az összes kliensnek, hogy tájékoztassa őket az új partner jelenlétéről. Azonban az új kliensnek küldött üzenet tartalmaz egy kiegészítő mezőt (amelyik az « az_öné » stringet tartalmazza) azért, hogy a partner tudja, hogy ez az üzenet az ő ágyújára vonatkozik és aktiválni tudja a megfelelő vezérlőpultot, miközben a server által hozzárendelt azonosítót tárolja (lásd a 35. - 37. sorokat is).

Következtetések és távlatok :

Ezt az alkalmazást didaktikai céllal mutattam be. Számos problémát szándékosan leegyszerűsítettem. Például, ha az olvasó maga teszteli ezeket a programokat, konstatálni fogja, hogy az üzenetek gyakran « csomagokba » vannak összefogva, ami szükségessé tenné az azok interpretálására készített algoritmusok finomítását. Alig vázoltam a játék alapműködését : a játékosok elosztását a két térfélen, a találatot kapott ágyúk eltüntetését, a különböző akadályokat, stb. Az olvasóra sok felfedeznivaló marad !

(18) Gyakorlatok :

18.1. Egyszerűsítse a 284 .oldalon leírt, « chat » kliensnek megfelelő scriptet úgy, hogy a két thread-objektum egyikét törli. Írja például úgy át, hogy a főthread-ben kezelje az üzenetek kibocsátását.

18.2. Módosítsa a 15. fejezet (229 oldal) játékát (egygépes változat) úgy, hogy csak egy ágyút és egy vezérlőpultot tartson meg. Adjon hozzá egy mozgó célpontot, aminek a mozgását egy független thread-objektum kezeli (olymódon, hogy a céltárgy és a lövedék animációját vezérlő kódrészek el legyenek választva).

18.8 Thread-ek (szálak) alkalmazása az animációk optimalizálására.

Az előző fejezet végén javasolt utolsó gyakorlat egy olyan alkalmazás fejlesztési módszert sugall, ami különösen érdekesnek bizonyulhat a több szimultán animációt tartalmazó videójátékok esetében.

Valóban : ha egy játék különböző animált elemeit mint a saját thread-jeikben működő független objektumokat programozzuk, akkor nemcsak a munkánkat egyszerűsítjük és javítjuk a scriptünk olvashatóságát, hanem a végrehajtás sebességét is növeljük és így ezeket az animációkat is folyamatosabbá tesszük. Ahhoz, hogy idáig eljussunk, le kell mondanunk az eddig használt késleltetési technikáról, de amit helyette fogunk alkalmazni végül is egyszerűbb !

18.8.1 Animációk késleltetése az after() segítségével

Az **after()** metódust (ez minden *Tkinter* widget-hez hivatalból van asszociálva) tartalmazó függvényből állt minden eddigi animáció « motorja ». Tudjuk, hogy ezzel a metódussal egy késleltetés vihető be a programunk menetébe : egy belső óra van aktiválva úgy, hogy megfelelő idő eltelte után a rendszer automatikusan hív valamilyen függvényt. Általában önmagát az **after()** -t tartalmazó függvényt hívjuk : így egy rekurzív hurkot állítunk elő, amiben a különböző grafikus objektumok elmozdulását kell programozni.

Programunk egyáltalán nem « fagy le », amíg az **after()** metódussal programozott időtartam nem telik le. Például ez alatt az időtartam alatt kattinthatunk egy gombra, átméretezhetjük az ablakot, a klaviatúráról adatot vihetünk be, stb. Ez hogyan lehetséges ?

Már többször említettem, hogy a modern grafikus alkalmazásoknak mindíg van egy motorja, ami egy folyamatosan a háttérben működő program : a főablak **mainloop()** metódusának aktiválásakor indul el. Mint a neve is jelzi, egy ugyanolyan típusú végtelen ciklust használ, mint a már jól ismert **while** ciklusok. Számos mechanizmus van beépítve ebbe a « motorba ». Az egyik a keletkező események fogadásából és abból áll, hogy azokat a programokat, amik kérik, utána megfelelő üzenetekkel figyelmezteti az események bekövetkeztére (lásd : *Eseményvezérelt programok* 85. oldal), mások a kiíratás szintjén végrehajtandó akciókat vezérlik, stb. Amikor egy widget

after () metódusát hívjuk, akkor valójában egy időmérő mechanizmust használunk, ami maga is bele van építve a **mainloop()**-ba és ennélfogva ez a központi adminisztrátor az, ami bizonyos idő eltelte után elindítja a kívánt függvény hívását.

Az **after()** metódust alkalmazó animációs technika az egyedüli lehetséges technika az egyetlen thread-en működő alkalmazások számára, mert kizárólag a **mainloop()** az, ami egy ilyen alkalmazás viselkedés együttesét vezérli. Nevezetesen : minden alkalommal a **mainloop()** felelős az ablak részleges vagy teljes újrarajzolásáért, ha az szükséges. Azért nem képzelhető el egy grafikus objektum koordinátáit átdefiniáló animációs motor megkonstruálása egy egyszerű **while** hurok belsejében, mert a **mainloop()** végrehajtása egész idő alatt fel lenne függesztve. Ennek az lenne a következménye, hogy ez alatt az idő alatt semmilyen objektum (speciálisan az, amit mozgatni szeretnénk !) sem lenne újrarajzolva. Úgy tűnne, hogy az egész alkalmazás mindaddig lefagyott, míg a **while** hurok meg nem szakad.

Mivel ez az egyetlen lehetőség, ezért a mono-thread alkalmazásainkban mindeddig ezt a technikát alkalmaztuk. Viszont ez egy zavaró nehézséggel jár : amiatt, hogy a **mainloop()** minden iterrációja nagyszámú műveletet kezel, ezért az **after()**-rel programozott késleltetés nem lehet nagyon rövid. Például ez a technika nem tud egy tipikus PC-n (*Pentium IV*, f = 1,5 GHz) 15 ms alá menni. Ezt a korlátot figyelembe kell venni, ha gyors animációkat akarunk fejleszteni.

Az **after()** metódussal kapcsolatos másik nehézség az animációs ciklus szerkezetében rejlik (tudni illik az egy « rekurziv » függvény vagy metódus, vagyis ami önmagát hívja) : nem mindíg egyszerű uralni az ilyen fajta logikai konstrukciót, különösen ha több, független grafikai objektum animációját akarjuk programozni, melyek számának és mozgásának változni kell az időben.

18.8.2 Animációk késleltetése a time.sleep() -pel

Eltekinthetünk a fent említett **after()** metódus korlátaitól, ha a grafikai objektumaink animációját független thread-ekre bízzuk. Ha így járunk el, megszabadulunk a **mainloop()** gyámságától és például a **while** vagy a **for** utasítás alkalmazásával a hagyományosabb ciklusszerkezeteken alapuló animációs eljárásokat konstruálhatunk.

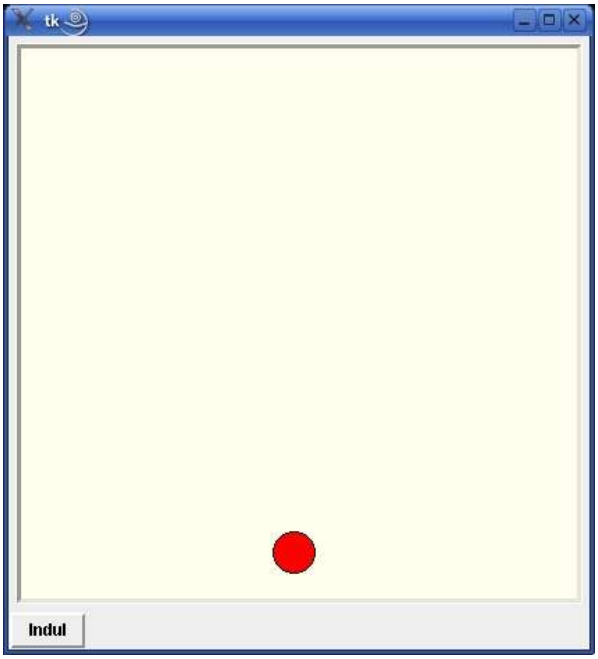
Viszont ügyelni kell rá, hogy mindegyik ciklus belsejébe szúrjunk be egy késleltetést, ami alatt « kezet nyújtunk » az operációs rendszernek (azért, hogy az más thread-ekkel is tudjon foglalkozni). Ennek megvalósításához a **time** modul **sleep()** függvényét fogjuk hívni. Ez a függvény teszi lehetővé az aktuális thread végrehajtásának felfüggesztését egy adott időtartamra, mialatt más thread-ek és alkalmazások folytatják a működésüket. Az így létrehozott késleltetés nem függ a **mainloop()**-tól, következésként sokkal rövidebb lehet, mint amit az **after()** metódus megenged.

Figyelem : ez nem jelenti azt, hogy maga a képernyőfrissítés gyorsabb lesz, mert azt továbbra is a **mainloop()** biztosítja. Viszont nagyon felgyorsíthatjuk azokat a mechanizmusokat, amiket mi magunk építünk be a saját animációs eljárásainkba. Például egy játékprogramban gyakori, hogy periódikusan össze kell hasonlítani két mozgó objektum (egy lövedék és egy céltárgy) pozícióját azért, hogy amikor érintkeznek el lehessen indítani egy akciót (robbanás, pontszámváltoztatás, stb.). Az itt leírt animációs technikával gyakrabban hajthatunk végre ilyen összehasonlításokat és így pontosabb eredményt remélhetünk. Ugyanígy, egy valósidejű pályaszámításnál növelhetjük a figyelembe vett pontok számát és így finomíthatjuk a pályát.

*Megjegyzés : Az **after()** metódus alkalmazásakor millisecundumokban, egész argumentumként kell megadni a kívánt késleltetést. A **sleep()** függvény hívásakor, az argumentumot secundumokban, float formában kell megadni. Viszont nagyon kis értékeket (például: 0.0003) használhatunk.*

18.8.3 Konkrét példa

Az alábbi script ennek a technikának az alkalmazását mutatja be egy szándékosan minimalista példán. Ez egy kis grafikus alkalmazás, amiben egy ábra mozog körbe egy vásznon. Az alkalmazás **mainloop()** « motorját », ahogy az megszokott, a főthread-ben elindítjuk el. Az alkalmazás constructora egy kör rajzát, egy gombot és egy thread-objektumot tartalmazó vásznat hoz létre. Ez az a thread-objektum, ami úgy biztosítja a rajz animációját, hogy semmilyen widget-nek sem hívja az **after()** metódusát. Ehelyett egy hagyományos ***while*** hurkot alkalmaz, ami a **run()** metódusában van installálva.




```
1.  from Tkinter import *
2.  from math import sin, cos
3.  import time, threading
4.
5.  class App(Frame):
6.      def __init__(self):
7.          Frame.__init__(self)
8.          self.pack()
9.          can =Canvas(self, width =400, height =400,
10.                     bg ='ivory', bd =3, relief =SUNKEN)
11.         can.pack(padx =5, pady =5)
12.         cercle = can.create_oval(185, 355, 215, 385, fill ='red')
13.         tb = Thread_labda(can, circle)
14.         Button(self, text ='Indul', command =tb.start).pack(side =LEFT)
15.         # Button(self, text ='Leáll', command =tb.stop).pack(side =RIGHT)
16.         # leállítjuk a másik thread-et, ha bezárjuk az ablakot :
17.         self.bind('<Destroy>', tb.stop)
18.
19. class Thread_labda(threading.Thread):
20.     def __init__(self, canevas, drawing):
21.         threading.Thread.__init__(self)
22.         self.can, self.drawing = canevas, drawing
23.         self.anim =1
24.
25.     def run(self):
26.         a = 0.0
27.         while self.anim == 1:
```

```
28.             a += .01
29.             x, y = 200 + 170*sin(a), 200 +170*cos(a)
30.             self.can.coords(self.dessin, x-15, y-15, x+15, y+15)
31.             time.sleep(0.010)
32.
33.     def stop(self, evt =0):
34.         self.anim =0
35.
36. App().mainloop()
```

Magyarázatok :

- 13. és 14. sor : Példánk maximális leegyszerűsítése érdekében az animációért felelős objektumot közvetlenül a főalkalmazás constructor-ában hozzuk létre. Ez a thread-objektum viszont csak akkor indul el, ha a felhasználó a « Marche » gombra kattint, ami a **start()** metódusát aktiválja (emlékezzünk rá, hogy ez az a beépített metódus, ami el fogja indítani a **run()** metódust, amiben az animációs hurkunk van).
- 15. sor : Egy befejezett thread-et nem indíthatunk újra. Ezért csak egyszer indíthatjuk el ezt az animációt (legalábbis az itt bemutatott formában). Hogy meggyőződjünk erről, távolítsuk el a15. sor elejéről a # karaktert (ami miatt a Python ezt a sort eddig kommentnek tekintette) : az animáció elindulásakor az egérrel a gombra kattintva előidézzük a 27. -31. sorok **while** ciklusából való kilépést, ami befejezi a **run()** metódust. Az animáció leáll, de az őt kezelő thread is befejeződik. Ha megpróbáljuk a « Indít» gombbal újraindítani a thread-et, csak egy hibaüzenetet kapunk.
- 26. - 31. sorok : Egyenletes körmozgás szimulációjához elég, ha folyamatosan változtatjuk az **a** szög értékét. Ennek a szögnek a sinus-ával és cosinus-ával kiszámolhatjuk a szögnek megfelelő kerületi pont **x** és **y** koordinátáit[80].
  A szög minden egyes iterrációnál csak egy század radiánt változik (körölbelül 0,6°), így 628 iterrációra van szükség, hogy a mozgó objektum egy teljes kört tegyen meg. Az iterráció számára választott késleltetés a 31. sorban található : 10 millisecundum. Ennek az értéknek a csökkentésével gyorsíthatjuk a mozgást, de nem mehetünk 1 millisecundum alá, ami már nem olyan rossz.

80 Erre vonatkozóan a 230 . oldalon található magyarázat .

19. Fejezet : Függelék

1.1 A Python telepítése

Ha ki akarja próbálni a Pythont a PC-jén, ne tétovázzon ! A telepítés egyszerű (és tökéletesen reverzibilis).

1.2 Telepítés Windows alatt

A Python hivatalos website-ján : **http://www.python.org** a « Download » fejezetben a különböző Python verziók telepítő programjai találhatók. Nyugodtan választhatja az utolsó « production » verziót.

Például, 2003.09.03-án ez a 2.3.1 verzió volt – A letöltenő file a: `Python-2.3.1.exe`

Másoljuk egy ideiglenes könyvtárba és hajtassuk végre. A Pythont alapértelmezetten egy « Python** » nevű könyvtárba lesz telepítve (a ** a verziószám két számjegyét jelenti) és az indító ikonok is automatikusan el leszen helyezve.

Amikor a telepítés befejeződött, törölhető az ideiglenes könyvtár tartalma.

1.3 Telepítés Linux alatt

Az olvasó valószínűleg egy olyan kereskedelmi disztró segítségével telepítette a Linux rendszerét, mint a *SuSE, RedHat* vagy *Mandrake.* Telepítse a Python package-et, ami a disztró részét képezi. Ne hagyja ki a *Tkinter*-t (ez néha a *Python imaging library*-vel egyidőben van telepítve).

1.4 Telepítés MacOS alatt

A Python-nak különböző verziói találhatók a MacOS 9 és Mac OS X -hez Jack Jansen website-ján : http://homepages.cwi.nl/~jack/macpython

Fontos megjegyzés a Python újabb verzióira vonatkozóan

A 2.3 verziójától kezdve a francia anyanyelvűeknek ajánlott az alábbi pseudo-comment-ek egyikét minden Python script elejére írni (az 1. vagy a 2. sorba) :

```
# -*- coding:Latin-1 -*-
```

vagy :

```
# -*- coding:Utf-8 -*-
```

A magyarázat a 40. oldalon található.

1.5 A SciTE (Scintilla Text Editor) telepítése

A SciTE egy kitűnő szövegszerkesztő, ami képes szintaktikai színezésre, automatikus kódkiegészítésre és *code folding-ra*, vagyis különböző utasításblokkok (egy osztály, egy függvény, egy ciklus, stb.) szándékos kimaszkolására.:Az utóbbi funkció akkor tűnik rendkívül praktikusnak, amikor a kód kezd hosszú lenni ... Egy terminálablakot is integrál, valamint a scriptek indításához egy shortcut-ot is egy.

Ez a szövegszerkesztő mind Windowshoz, mind Linuxhoz rendelkezésre áll.

Nézze meg a : http://www.scintilla.org/SciTE.html website-ot.

1.5.1 Telepítés Linux alatt :

A Scintilla szövegszerkesztőt a jelenlegi Linux disztribúciókkal hivatalosan együtt adják. Ha nincs a disztróban, akkor a fent említett website-ról le lehet tölteni, majd :

- Le kell tölteni a **gscite***.tgz** archív file-t, majd ki kell csomagolni a tar -ral.
- A **SciTE** végrehajtható állományt a **/usr/local/bin** -be kell telepíteni.
- Az összes többi állományt (*.properties file-okat) a **/usr/share/scite** -be (nem pedig a / usr/share/gscite -be!) kell telepíteni.

1.5.2 Telepítés Windows alatt :

- Le kell tölteni a **wscite***.zip** archív file-t, majd ki kell tömöríteni a \Program files -ba
- Egy indító ikont kell installálni a SciTe.exe -nek

1.5.3 A két verzióhoz :

Sokmindent testre lehet szabni (fontok, stb.) a globális tulajdonságok file-jának szerkesztésével (Menu Options → Open global options file).

Például a code folding balmargón lévő + és – szimbólumainak aktiválásához :

```
fold.symbols = 2            # a bekeretezett + és – ikonokhoz
fold.on.open = 1            # kiinduláskor minden be van csomagolva
margin.width =0             # a haszontalan margók törlése
```

Az automatikus tabulálás 4 betűközzel való helyettesítéséhez :

```
tabsize = 4
indent.size = 4
use.tabs = 0
```

1.6 A Python mega-widgetek telepítése

Látogassa meg a : http://pmw.sourceforge.net website-ot.

Kattintson a : **Download Pmw12tar.gz** linkre a megfelelő file letöltéséhez.

Fejtse ki ezt az archív filet egy ideiglenes könyvtárba egy olyan kitömörítő programmal, mint pl. a : *tar, Winzip, Info-Zip*, *unzip* ....

Másolja az automatikusan létrehozott **Pmw** alkönyvtárat abba a könyvtárba, ahová a Pythont telepíttette.

Windows alatt, ez például a `C:\Python23`

Linux alatt, valószínűleg ez a : `/usr/lib/python`

1.7 A Gadfly telepítése (adatbázisrendszer)

Töltsük le a http://sourceforge.net/projects/gadfly website-ról a **gadfly-1.0.0.tar.gz** csomagot. Ez egy tömörített archív file. Másoljuk ezt a file-t egy ideiglenes könyvtárba.

1.8 Telepítés Windows alatt :

Egy tetszőleges ideiglenes könyvtárban tömörítsük ki az archív file-t egy olyan programmal, mint a *Winzip*.

Nyissunk meg egy DOS ablakot és lépjünk be az automatikusan létrehozott alkönyvtárba.

Indítsuk el a : `python setup.py install` parancsot. Ez minden.

Esetleg javíthatjuk a teljesítményt a következő művelettel :

Az automatikusan létrehozott alkönyvtárban nyissuk meg a **kjbuckets** alkönyvtárat, majd nyissuk meg a Python verziónknak megfelelő alkönyvtárat. Másoljuk az ott található *.pyd file-t a Python telepítésünk gyökérkönyvtárába.

Amikor végeztünk, töröljük az ideiglenes könyvtárunkat.

1.8.1 Telepítés Linux alatt :

Adminisztrátorként (root), válasszunk valamilyen ideiglenes könyvtárat és tömörítsük oda ki a ***tar*** utility-vel, ami biztosan része a disztribúciónknak, az archív file-t.
Írjuk be a : `tar -xvzf gadfly-1.0.0.tar.gz`

Lépjünk be az automatikusan létrehozott alkönyvtárba: `cd gadfly-1.0.0`

Indítsuk el a : `python setup.py install` parancsot. Ez minden.

Ha a Linux rendszerünknek van C fordítója, akkor a kjbuckets könyvtár újrafordításával javíthatjuk a Gadfly teljesítményét. Ehhez írjuk még be a következő két utasítást :

```
cd kjbuckets
python setup.py install
```

Amikor minden befejeződött, töröljük az ideiglenes könyvtár tartalmát.

1.9 A gyakorlatok megoldásai

Néhány gyakorlatnak nem adom meg a megoldását. Törekedjen egyedül megtalálni a megoldást még akkor is, ha az nehéznek tűnik. Az olvasó akkor tanul a legtöbbet, ha sajátmaga oldja meg ezeket a problémákat.

4.2 gyakorlat :

```
>>> c = 0
>>> while c < 20:
...     c = c +1
...     print c, "x 7 =", c*7
```

vagy :

```
>>> c = 1
>>> while c <= 20:
...     print c, "x 7 =", c*7
...     c = c +1
```

4.3 gyakorlat :

```
>>> s = 1
>>> while s <= 16384:
...     print s, "euro =", s *1.65, "dollar"
...     s = s *2
```

4.4 gyakorlat :

```
>>> a, c = 1, 1
>>> while c < 13:
...     print a,
...     a, c = a *3, c+1
```

4.6 gyakorlat :

```
# Kezdő értékként megadott másodpercek száma :
# (egy nagy számot adunk meg !)
nsd = 12345678912
# Egy napra eső másodpercek száma :
nspj = 3600 * 24
# Egy évre eső másodpercek száma (365 napra -
# a szökőéveket nem vesszük figyelembe ) :
nspa = nspj * 365
# Egy hónapra eső másodpercek száma (feltételezzük,
# hogy minden hónap 30 napos) :
nspm = nspj * 30
# A megadott időtartam ennyi évet tesz ki :
na = nsd / nspa          # egészosztás
nsr = nsd % nspa         # a maradék másodpecek száma
# A megmaradó hónapok száma :
nmo = nsr / nspm         # egészosztás
nsr = nsr % nspm         # a maradék másodpecek száma
# A megmaradó napok száma :
nj = nsr / nspj          # egészosztás
nsr = nsr % nspj         # a maradék másodpecek száma
# A megmaradó órák száma :
nh = nsr / 3600          # egészosztás
nsr = nsr % 3600         # a maradék másodpecek száma
# a maradék pecek száma :
```

```
nmi = nsr /60           # egészosztás
nsr = nsr % 60          # a maradék másodpecek száma

print "Az átalakítandó másodpercek száma :", nsd
print "Ez az időtartam", na, "évnek"
print nmo, "hónapnak",
print nj, "napnak,",
print nh, "órának,",
print nmi, "percnek és",
print nsr, " másodpercnek felel meg."
```

4.7 gyakorlat :

```
# A 7-es szorzótábla első elemének kiíratása,
# 3 többszöröseinek jelzésével :

i = 1               # számláló: 1-től 20-ig egymás után vesszük az értékeket
while i < 21:
    # a kiírandó szorzat kiszámolása :
    t = i * 7
    # sorugrás néküli kiíratás (a vessző hasznáata) :
    print t,
    # a szorzat 3 többszöröse ? (a modulo operátor alkalmazása) :
    if t % 3 == 0:
        print "*",      # ez esetben kiíratunk egy csillagot
    i = i + 1           # minden esetben incrementáljuk a számlálót
```

5.1 gyakorlat :

```
# fok -> radián átalakítás
# Ismétlés : 1 radiános az a szög, melyhez tartozó körív hossza
# a kör sugarának hosszával.
# Mivel a kerület 2 pi R, ezért egy 1 radiános szög
# 360° / 2 pi -nek, vagy 180° / pi -nek felel meg

# A kiindulási szög fok, perc, másodpercben megadva :
deg, min, sec  = 32, 13, 49

# A szögmásodpercek átalakítása szögpercekbe :
# (a tizedespont miatt az átalakítás eredménye valós szám lesz)
fm = sec/60.
# A szögpercek átalakítása fokokká :
fd = (min + fm)/60
# A szög értékének tizedestörtté alakítása :
ang = deg + fd
# pi értéke :
pi = 3.14159265359
# 1 radián fokokban megadva :
rad = 180 / pi
#  A szög átalakítása radiánná :
arad = ang / rad
# Kiírás :
print deg, "°", min, "'", sec, '" =', arad, "radián"
```

5.3 gyakorlat:

```
# °Fahrenheit <-> °Celsius átalakítás

# A) °C-ban megadott hőmérséklet :
tempC = 25
# Átalakítás °Fahrenheitbe :
tempF = tempC * 1.8 + 32
# Kiíratás :
print tempC, "°C =", tempF, "°F"

# B) °F-ben megadott hőmérséklet :
tempF = 25
# Átalakítás °Celsiusba :
tempC = (tempF - 32) / 1.8
# Kiíratás :
print tempF, "°F =", tempC, "°C"
```

5.5 gyakorlat:

```
>>> a, b = 1, 1                  # változat :  a, b = 1., 1
>>> while b<65     :
...     print b, a
...     a,b = a*2, b+1
...
```

5.6 gyakorlat:

```
# Megadott karakter keresése egy stringben

# A kiindulásul megadott string :
ch = "Monty python flying circus"
# A keresendő karakter :
cr = "e"
# Keresés :
lc = len(ch)    # az ellenőrzendő karakterek száma
i = 0           # a karakter indexe a vizsgálat során
t = 0           # a beállítandó"flag", ha a keresett karakter megvan
while i < lc:
    if ch[i] == cr:
        t = 1
    i = i + 1
# Kiíratás :
print "A karakter", cr,
if t == 1:
    print "megvan",
else:
    print "nem található meg",
print "a stringben", ch
```

5.8 gyakorlat:

```
# Helykitöltő karakter beszúrása egy stringbe

# A kiindulási string :
ch = "Gaston"
# A beszúrandó karakter :
cr = "*"
# A beszúrandó karakterek száma eggyel kisebb a stringben lévõ karakterek
```

```
# számánál. A stringet a második karakterétõl kezdve manipuláljuk
# (nem vesszük figyelembe az elsõ karaktert).
lc = len(ch)    # az összes karakter száma
i = 1           # az első vizsgálandó karakter indexe (a második valójában)
nch = ch[0]     # a létrehozandó új string (már tartalmazza az első karaktert.)
while i < lc:
    nch = nch + cr + ch[i]
    i = i + 1
# Kiíratás :
print nch
```

5.9 gyakorlat:

```
# String inverziója

# Kiindulási string :
ch = "zorglub"
lc = len(ch)    # az összes karakter száma
i = lc - 1      # az utolsó karaktertől fogunk kezdeni
nch = ""        # a létrehozandó új string (kezdetben üres)
while i >= 0:
    nch = nch + ch[i]
    i = i - 1
# Kiíratás :
print nch
```

5.11 gyakorlat:

```
# Két lista átalakítása  egy listává

# A kiindulási listák :
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Január','Február','Március','Április','Május','Június',
      'Július','Augusztus','Szeptember','Októbrer','November','December']
# A létrehozandó új lista (kezdetben üres) :
t3 = []
# Ciklus :
i = 0
while i < len(t1):
    t3.append(t2[i])
    t3.append(t1[i])
    i = i + 1

# Kiíratás :
print t3
```

5.12 gyakorlat:

```
# Lista elemeinek kiíratása

# A kiindulási lista :
t2 = ['Január','Február','Március','Április','Május','Június',
      'Július','Augusztus','Szeptember','Októbrer','November','December']
# Kiíratás :
i = 0
while i < len(t2):
    print t2[i],
    i = i + 1
```

5.13 gyakorlat:

```
# Lista legnagyobb elemének megkeresése

# A kiindulási lista :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
# A lista kezelése közben az alábbi változóban fogjuk tárolni a
# már megtalált legnagyobb elemet :
max = 0
# Az összes elem vizsgálata :
i = 0
while i < len(tt):
    if tt[i] > max:
        max = tt[i]         # egy új maximális érték tárolása
    i = i + 1
# Kiíratás :
print "A lista legnagyobb elemének az értéke", max
```

5.14 gyakorlat:

```
# A páros és páratlan számok szétválasztása

# A kiindulási lista :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
paros = []
paratlan = []
# Az összes elem vizsgálata :
i = 0
while i < len(tt):
    if tt[i] % 2 == 0:
        paros.append(tt[i])
    else:
        paratlan.append(tt[i])
    i = i + 1
# Kiíratás :
print "Páros számok :", paros
print "Páratlan számok :", paratlan
```

6.1 gyakorlat:

```
# mérföld/óra átalakítása km/h -vá és m/s -má

print "Írja be az óránként megtett mérföldek számát : ",
ch = raw_input()            # általában az input() a preferált
mph = float(ch)             # a bemeneti string átalakítása valós számmá
mps = mph * 1609 / 3600     # átalakítás m/s -má
kmph = mph * 1.609          # átalakítás km/h -vá
# Kiíratás :
print mph, " mérföld/óra  =", kmph, "km/h, vagy", mps, "m/s"
```

6.2 gyakorlat:

```
# Tetszőleges háromszög kerülete és területe

from math import sqrt

print "Írja be az a oldalt : "
a = float(raw_input())
print " Írja be a b oldalt : "
b = float(raw_input())
```

```
print " Írja be a c oldalt  : "
c = float(raw_input())
d = (a + b + c)/2                # a kerület fele
s = sqrt(d*(d-a)*(d-b)*(d-c))    # terület (a képlet alapján)

print "Az oldalak hossza =", a, b, c
print "Kerület =", d*2, "Terület =", s
```

6.4 gyakorlat:

```
# Elemek beírása egy listába

tt = []             # Az elkészítendő lista (kezdetben üres)
ch = "start"        # valamilyen érték (de nem nulla)
while ch != "":
    print "Írjon be egy értéket : "
    ch = raw_input()
    if ch != "":
        tt.append(float(ch))        # másik változat : tt.append(ch)

# a lista kiíratása :
print tt
```

6.8 gyakorlat:

```
# Két határérték közé eső egész számok kezelése

print "Írja be az alsó határt :",
a = input()
print " Írja be a felső határt :",
b = input()
s = 0                   # a keresett összeg (kezdetben nulla)
# Az a és b közé eső számsorozat bejárása (a és b -t is beleértve) :
n = a                   # az aktuálisan kezelt szám
while n <= b:
    if n % 3 ==0 and n % 5 ==0:      # változat : 'or' az 'and' helyett
        s = s + n
    n = n + 1

print "A keresett összeg", s
```

6.9 gyakorlat:

```
# Szököévek

print "Írja be az ellenőrzendő évszámot :",
a = input()

if a % 4 != 0:
    # a nem osztható 4-gyel -> nem szökőév
    bs = 0
else:
    if a % 400 ==0:
        # a osztható 400-zal -> szökőév
        bs = 1
    elif a % 100 ==0:
        # a osztható 100-zal -> nem szökőév
        bs = 0
    else:
```

```
        # más esetek, amikor a osztható 4-gyel -> szökőév
        bs = 1
if bs ==1:
    ch = ""
else:
    ch = "nem"
print a, ch, " szökőév."
```

(Alex Misbah által javasolt változat) :

```
a=input('Írjon be egy évszámot:')

if (a%4==0) and ((a%100!=0) or (a%400==0)):
    print a,"szökőév"
else:
    print a,"nem szökőév"
```

6.11 gyakorlat : Háromszögek számolásai

```
from sys import exit      # rendszerfüggvényeket tartalmazó  modul

print """
Írja be a három oldal hosszát
(az értékeket vesszővel válassza el) :"""
a, b, c = input()
# Nem lehet olyan háromszöget konstruálni, melynek minden oldala
# rövidebb, mint a másik két oldal hosszának összege :
if a < (b+c) and b < (a+c) and c < (a+b) :
    print "Ez a három hossz egy háromszöget definiál."
else:
    print "Nem lehet ilyen háromszöget konstruálni !"
    exit()          # ezért kilépünk a programból.

f = 0
if a == b and b == c :
    print "Ez a háromszög egyenlőoldalú."
    f = 1
elif a == b or b == c or c == a :
    print "Ez a háromszög egyenlőszárú."
    f = 1
if a*a + b*b == c*c or b*b + c*c == a*a or c*c + a*a == b*b :
    print "Ez a háromszög derékszögű."
    f = 1
if f == 0 :
    print " Ez egy általános háromszög."
```

6.15 gyakorlat :

```
# Iskolai érdemjegyek
jegyek = []           # A létrehozandó lista
n = 2                # valamilyen pozitív érték a ciklus elindításához
while n >= 0 :
    print "Írjon be egy érdemjegyet. : ",
    n = float(raw_input())      # egész számmá alakítás
    if n < 0 :
        print "OK. Befejezés."
    else:
        jegyek.append(n)         # hozzátesz egy érdemjegyet a listához
        # A már be beírt érdemjegyeken végzett különböző számítások  :
        # minimális és maximális értékek + a jegyek összege.
        min = 500               # valamennyi érdemjegynél nagyobb érték
        max, tot, i = 0, 0, 0
        nn = len(jegyek)         # A már beírt érdemjegyek száma
        while i < nn:
            if jegyek[i] > max:
                max = jegyek[i]
            if jegyek[i] < min:
                min = jegyek[i]
            tot = tot + jegyek[i]
            atlag = tot/nn
            i = i + 1
        print "Jegyek=", nn, "Max =", max, "Min =", min, "Átl. =", atlag
```

7.3 gyakorlat :

```
from math import pi

def surfCircle(r):
    "Egy r sugarú kör területe"
    return pi * r**2

# teszt :
print surfCircle(2.5)
```

7.4 gyakorlat :

```
def volBox(x1, x2, x3):
    "Egy parallelepipedon térfogata"
    return x1 * x2 * x3

# teszt :
print volBox(5.2, 7.7, 3.3)
```

7.5 gyakorlat :

```
def maximum(n1, n2, n3):
    "Renvoie le plus grand de trois nombres"
    if n1 >= n2 and n1 >= n3:
        return n1
    elif n2 >= n1 and n2 >= n3:
        return n2
    else:
        return n3

# teszt :
print maximum(4.5, 5.7, 3.9)
```

7.9 gyakorlat:

```
def karSzamlalo(ca, ch):
    "Megadja a ca karakter elõfordulásainak számát a ch stringben"
    i, tot = 0, 0
    while i < len(ch):
        if ch[i] == ca:
            tot = tot + 1
        i = i + 1
    return tot

# teszt :
print karSzamlalo("e","Ez a karakterlánc egy példa")
```

7.10 gyakorlat:

```
def indexMax(tt):
    "a tt lista legnagyobb elemének indexét adja meg"
    i, max = 0, 0
    while i < len(tt):
        if tt[i] > max :
            max, imax = tt[i], i
        i = i + 1
    return imax

# teszt :
serie = [5, 8, 2, 1, 9, 3, 6, 4]
print indexMax(serie)
```

7.11 gyakorlat:

```
def honapNev(n):
    "az év n-edik hónapjának nevét adja meg"
    mois = ['Január','Február','Március','Április','Május','Június','Július',
            'Augusztus','Szeptember','Október','November','December']
    return mois[n -1]       # az indexek számozása nullával kezdõdik

# teszt :
print honapNev(4)
```

7.14 gyakorlat:

```
def volBox(x1 =10, x2 =10, x3 =10):
    "Egy  parallelepipedon térfogata"
    return x1 * x2 * x3

# teszt :
print volBox()
print volBox(5.2)
print volBox(5.2, 3)
```

7.15 gyakorlat:

```
def volBox(x1 =-1, x2 =-1, x3 =-1):
    "Egy parallelepipedon térfogata"
    if x1 == -1 :
        return x1           # nem adtunk meg argumentumot
    elif x2 == -1 :
        return x1**3        # egy argumentum -> kocka
    elif x3 == -1 :
        return x1*x1*x2     # két argumentum -> prizmatikus test
    else :
        return x1*x2*x3

# teszt :
print volBox()
print volBox(5.2)
print volBox(5.2, 3)
print volBox(5.2, 3, 7.4)
```

7.16 gyakorlat:

```
def changeChar(ch, ca1, ca2, eleje =0, vege =-1):
    "A ch karakterláncban kicseréli az összes ca1 karaktert a ca2 karakterre"
    if vege == -1:
        vege = len(ch)
    nch, i = "", 0            # nch : a létrehozandó új karakterlánc
    while i < len(ch) :
        if i >= eleje and i <= vege and ch[i] == ca1:
            nch = nch + ca2
        else :
            nch = nch + ch[i]
        i = i + 1
    return nch

# teszt :
print changeChar("Ceci est une toute petite phrase", " ", "*")
print changeChar("Ceci est une toute petite phrase", " ", "*", 8, 12)
print changeChar("Ceci est une toute petite phrase", " ", "*", 12)
```

7.17 gyakorlat:

```
def eleMax(lst, eleje =0, vege =-1):
    "Az lst legnagyobb elemét adja meg"
    if vege == -1:
        vege = len(lst)
    max, i = 0, 0
    while i < len(lst):
        if i >= eleje and i <= vege and lst[i] > max:
            max = lst[i]
        i = i + 1
    return max

# teszt :
serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
print eleMax(serie)
print eleMax(serie, 2)
print eleMax(serie, 2, 5)
```

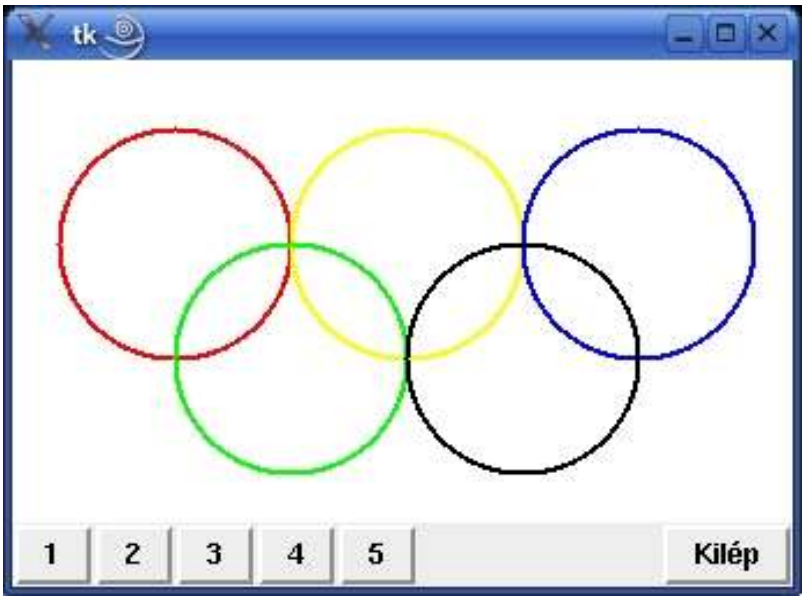
8.7 gyakorlat:

```
# Olimpiai karikák  - tömör változat.

from Tkinter import *

# Az 5 karika X,Y koordinátái :
coord = [[20,30], [120,30], [220, 30], [70,80], [170,80]]
# Az 5 karika színei :
colour = ["red", "yellow", "blue", "green", "black"]

base = Tk()
can = Canvas(base, width =335, height =200, bg ="white")
can.pack()
bou = Button(base, text ="Kilép", command =base.quit)
bou.pack(side = RIGHT)
# Az 5 karika rajza :
i = 0
while i < 5:
    x1, y1 = coord[i][0], coord[i][1]
    can.create_oval(x1, y1, x1+100, y1 +100, width =2, outline =colour[i])
    i = i +1
base.mainloop()
```

Változat:

```
# Olimpiai karikák  - minden karikát külön rajzoló változat.

from Tkinter import *

# Függvények az 5 karika rajzolásához :

def drawCircle(i):
    x1, y1 = coord[i][0], coord[i][1]
    can.create_oval(x1, y1, x1+100, y1 +100, width =2, outline =colour[i])

def a1():
    drawCircle(0)

def a2():
    drawCircle(1)

def a3():
    drawCircle(2)

def a4():
    drawCircle(3)

def a5():
    drawCircle(4)

##### Fõprogram : ##########

# Az 5 karika X,Y koordinátái :
coord = [[20,30], [120,30], [220, 30], [70,80], [170,80]]
# az 5 karika színei :
colour = ["red", "yellow", "blue", "green", "black"]

base = Tk()
can = Canvas(base, width =335, height =200, bg ="white")
can.pack()
```

```
but = Button(base, text ="Kilép", command =base.quit)
but.pack(side = RIGHT)

# Az 5 gomb létrehozása :
Button(base, text='1', command = a1).pack(side =LEFT)
Button(base, text='2', command = a2).pack(side =LEFT)
Button(base, text='3', command = a3).pack(side =LEFT)
Button(base, text='4', command = a4).pack(side =LEFT)
Button(base, text='5', command = a5).pack(side =LEFT)
base.mainloop()
```



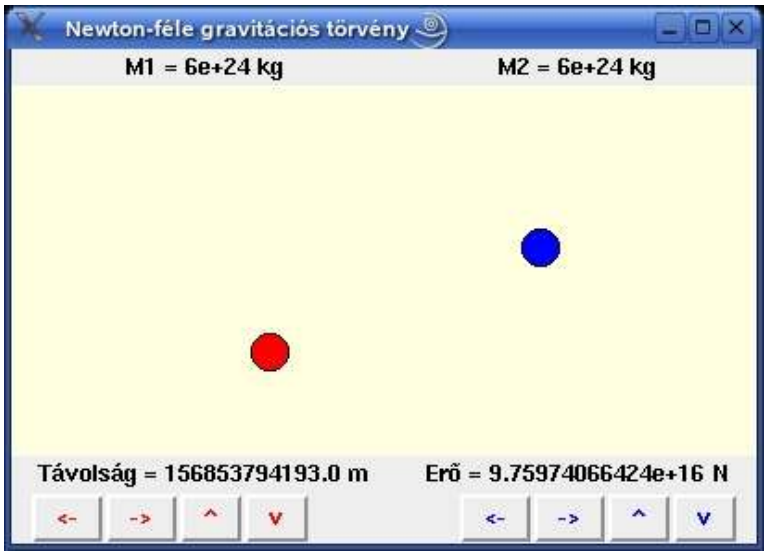
8.9 és 8.10 gyakorlat :

```
# Dámatábla véletlenszerűen elhelyezett korongokkal

from Tkinter import *
from random import randrange        # véletlenszám generátor

def dama_tabla():
    "10 négyzetből álló sor rajzolása váltakozó eltolódással"
    y = 0
    while y < 10:
        if y % 2 == 0:              # két alkalomból egyszer
            x = 0                   # a négyzetekből álló sor
        else:                       # egy négyzetnyi
            x = 1                   # eltolódással fog kezdődni
        line_of_squares(x*c, y*c)
        y += 1

def line_of_squares(x, y):
    "x, y -ból kiindulva négyzetekből álló vonal rajzolása"
    i = 0
    while i < 10:
        can.create_rectangle(x, y, x+c, y+c, fill='navy')
        i += 1
        x += c*2                    # négyzetes közök hagyása

def circle(x, y, r, colour):
    "x,y középpontú és r sugarú kör rajzolása"
    can.create_oval(x-r, y-r, x+r, y+r, fill=colour)

def korong_hozzaadasa():
    "korong véletlenszerű rajzolása a dámatáblára"
    # a korong coordinátáinak sorsolása :
    x = c/2 + randrange(10) * c
    y = c/2 + randrange(10) * c
    circle(x, y, c/3, 'red')

##### Főprogram : ############

# Próbálja meg jól "paraméterezni" a programjait, ahogyan ebben a scriptben
tettük.
# A script tetszőleges méretű dámatáblákat rajzolhat egyetlen érték,
# a négyzet méretének megváltoztatásával :

c = 30                  # négyzetek mérete

ablak = Tk()
can = Canvas(ablak, width =c*10, height =c*10, bg ='ivory')
can.pack(side =TOP, padx =5, pady =5)
b1 = Button(ablak, text ='dámatábla', command =dama_tabla)
b1.pack(side =LEFT, padx =3, pady =3)
b2 = Button(ablak, text ='korongok', command =korong_hozzaadasa)
b2.pack(side =RIGHT, padx =3, pady =3)
ablak.mainloop()
```

8.12 gyakorlat:

```
# Gravitáció szimulálása

from Tkinter import *
from math import sqrt

def distance(x1, y1, x2, y2):
    "az x1,y1 és x2,y2 pontokat elválasztó távolság"
    d = sqrt((x2-x1)**2 + (y2-y1)**2)       # Pythagoras tétele
    return  d

def forceG(m1, m2, di):
       "egymástól di távolságra lévõ m1 et m2 tömegû tömegpontok közötti
gravitációs erõ"
    return m1*m2*6.67e-11/di**2             # Newton-törvény

def mozdul(n, gd, hb):
    "az n csillag elmozdulása, balról jobbra vagy fentrõl lefelé"
    global x, y, step
    # új koorddináták :
    x[n], y[n] = x[n] +gd, y[n] +hb
    # a rajz elmozdulása a vásznon :
    can.coords(astre[n], x[n]-10, y[n]-10, x[n]+10, y[n]+10)
    # az új távolság számolása :
    di = distance(x[0], y[0], x[1], y[1])
    # a "képernyõtávolság" konverziója "asztronómiai távolsággá" :
    diA = di*1e9            # (1 pixel => 1 millió km)
    # a megfelelõ gravitációs erõ számolása :
    f = forceG(m1, m2, diA)
    # az új távolság- és erõértékek számolása :
    valDis.configure(text="Távolság = " +str(diA) +" m")
    valFor.configure(text="Erõ = " +str(f) +" N")
    # a "lépés" adaptálása a távolság függvényében :
    step = di/10

def balra1():
    mozdul(0, -step, 0)

def jobbra1():
    mozdul(0, step, 0)

def fel1():
    mozdul(0, 0, -step)

def le1():
    mozdul(0, 0, step)

def balra2():
    mozdul(1, -step, 0)

def jobbra2():
    mozdul (1, step, 0)

def fel2():
    mozdul(1, 0, -step)

def le2():
    mozdul(1, 0, step)

# A csillagok tömege :
m1 = 6e24          # (a Föld tömege, kg-ban)
```

```
m2 = 6e24           #
astre = [0]*2       # lista a rajzok hivatkozásainak tárolására
x =[50., 350.]      # a csillagok X (képernyõ)koordinátáinak  listája
y =[100., 100.]     # a csillagok Y (képernyõ)koordinátáinak  listája
step =10            # egy elmozdulás "lépés"

# Ablak létrehozása :
ablak = Tk()
ablak.title(' Newton-féle gravitációs törvény')
# Címkék :
valM1 = Label(ablak, text="M1 = " +str(m1) +" kg")
valM1.grid(row =1, column =0)
valM2 = Label(ablak, text="M2 = " +str(m2) +" kg")
valM2.grid(row =1, column =1)
valDis = Label(ablak, text="Distance")
valDis.grid(row =3, column =0)
valFor = Label(ablak, text="Force")
valFor.grid(row =3, column =1)
# Vászon 2 csillag rajzával:
can = Canvas(ablak, bg ="light yellow", width =400, height =200)
can.grid(row =2, column =0, columnspan =2)
astre[0] = can.create_oval(x[0]-10, y[0]-10, x[0]+10, y[0]+10,
                           fill ="red", width =1)
astre[1] = can.create_oval(x[1]-10, y[1]-10, x[1]+10, y[1]+10,
                           fill ="blue", width =1)
# 4 gombból álló 2 gombcsoport, mindegyikük egy keretbe (frame) van téve :
fra1 = Frame(ablak)
fra1.grid(row =4, column =0, sticky =W, padx =10)
Button(fra1, text="<-", fg ='red',command =balra1).pack(side =LEFT)
Button(fra1, text="->", fg ='red', command =jobbra1).pack(side =LEFT)
Button(fra1, text="^", fg ='red', command =fel1).pack(side =LEFT)
Button(fra1, text="v", fg ='red', command =le1).pack(side =LEFT)
fra2 = Frame(ablak)
fra2.grid(row =4, column =1, sticky =E, padx =10)
Button(fra2, text="<-", fg ='blue', command =balra2).pack(side =LEFT)
Button(fra2, text="->", fg ='blue', command =jobbra2).pack(side =LEFT)
Button(fra2, text="^", fg ='blue', command =fel2).pack(side =LEFT)
Button(fra2, text="v", fg ='blue', command =le2).pack(side =LEFT)

ablak.mainloop()
```



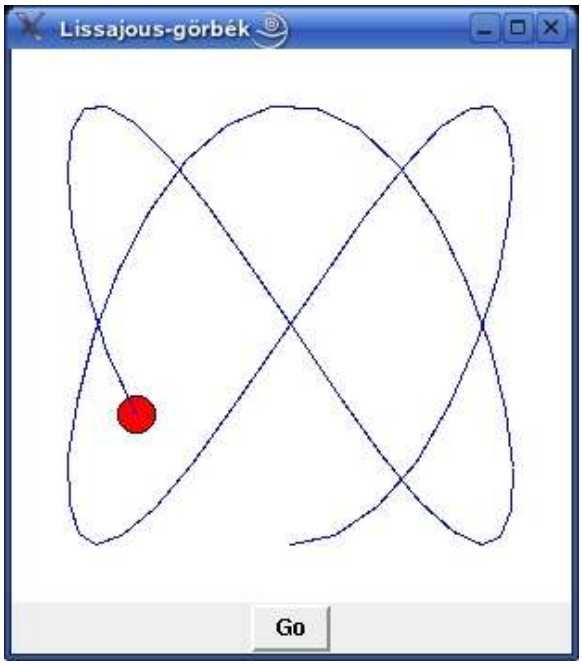
8.16 gyakorlat :

```
# Fahrenheit <=> Celsius átalakítás

from Tkinter import *

def convFar(event):
    "a hõmérséklet értéke Fahrenheit fokban kifejezve"
    tF = eval(mezoTC.get())
    varTF.set(str(tF*1.8 +32))

def convCel(event):
    "a hõmérséklet értéke Celsius fokban kifejezve"
    tC = eval(mezoTF.get())
    varTC.set(str((tC-32)/1.8))

ablak = Tk()
ablak.title('Fahrenheit/Celsius')

Label(ablak, text='Temp. Celsius :').grid(row =0, column =0)
# Adatbeviteli mezõhöz asszociált "Tkinter változó". Ez az "objektum-változó"
# biztosítja az interface-t a TCL és a Python között (ld. jegyzet, 165. oldal) :
varTC =StringVar()
mezoTC = Entry(ablak, textvariable =varTC)
mezoTC.bind("<Return>", convFar)
mezoTC.grid(row =0, column =1)
# A Tkinter változó tartalmának inicializálása :
varTC.set("100.0")

Label(ablak, text='Temp. Fahrenheit :').grid(row =1, column =0)
varTF =StringVar()
mezoTF = Entry(ablak, textvariable =varTF)
mezoTF.bind("<Return>", convCel)
mezoTF.grid(row =1, column =1)
varTF.set("212.0")

ablak.mainloop()
```

8.18 és 8.20 gyakorlat :

```
# Körök és  Lissajous-görbék

from Tkinter import *
from math import sin, cos

def move():
    global ang, x, y
    # az elõzõ koordinátákat tároljuk az újak számolása elõtt :
    xp, yp = x, y
    # elforgatás 0.1 radiánnal :
    ang = ang +.1

    # a szög sinus-a és cosinus-a => trigon. kör egy pontjának koordinátái.
```

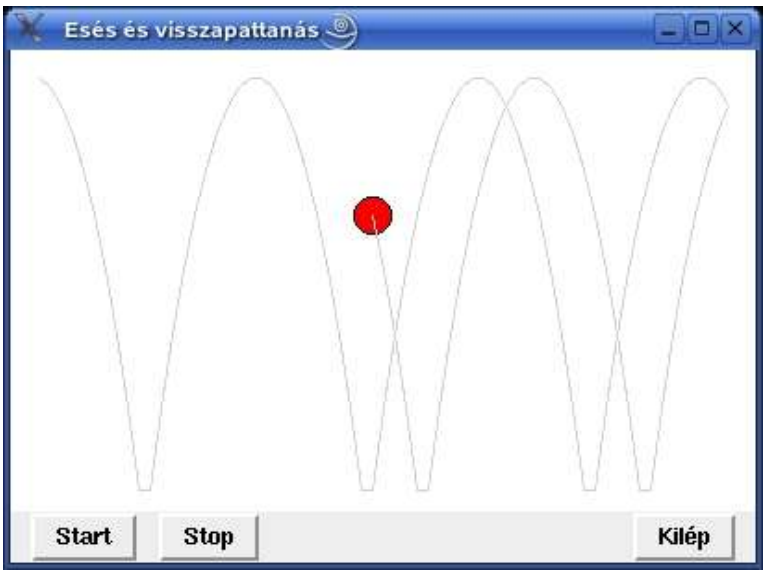
```
    x, y = sin(ang), cos(ang)
    # A Lissajous görbét f1/f2 = 2/3 -vel definiáló változat:
    # x, y = sin(2*ang), cos(3*ang)
    # skálázás (120 = kör sugara, (150,150) = vászon közepe
    x, y = x*120 + 150, y*120 + 150
    can.coords(labda, x-10, y-10, x+10, y+10)
    # can.create_line(xp, yp, x, y, fill ="blue")       # pályarajzolás
ang, x, y = 0., 150., 270.
ablak = Tk()
ablak.title('Lissajous-görbék')
can = Canvas(ablak, width =300, height=300, bg="white")
can.pack()
labda = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(ablak, text='Go', command =move).pack()

ablak.mainloop()
```



8.27 gyakorlat:

```
# Esés és visszapattanás

from Tkinter import *

def move():
    global x, y, v, dx, dv, flag
    xp, yp = x, y            # az elõzõ koordináták tárolása
    # vízszintes elmozdulás :
    if x > 385 or x < 15 :   # visszapattanás az oldalfalról :
        dx = -dx             # invertáljuk az elmozdulást
    x = x + dx
    # a függõleges sebesség variációja (mindíg lefelé):
    v = v + dv
    # függõleges elmozdulás (arányos a sebességgel)
    y = y + v
    if y > 240:              # a föld szintje 240 pixelre :
        y = 240              # nem mehet tovább !
```

```
            v = -v                  # visszapattan : a sebesség megfordul
        # újra pozícionáljuk a labdát :
        can.coords(labda, x-10, y-10, x+10, y+10)
        # pályavéget rajzolunk :
        can.create_line(xp, yp, x, y, fill ='light grey')
        # ... addig csináljuk, amíg szükséges :
        if flag > 0:
            ablak.after(50,move)

def start():
    global flag
    flag = flag +1
    if flag == 1:
        move()

def stop():
    global flag
    flag =0

# a koordináták, a sebességek és az animációflag inicializálása :
x, y, v, dx, dv, flag  = 15, 15, 0, 6, 5, 0

ablak = Tk()
ablak.title(' Esés és visszapattanás')
can = Canvas(ablak, width =400, height=250, bg="white")
can.pack()
labda = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(ablak, text='Start', command =start).pack(side =LEFT, padx =10)
Button(ablak, text='Stop', command =stop).pack(side =LEFT)
Button(ablak, text='Kilép', command =ablak.quit).pack(side =RIGHT, padx =10)

ablak.mainloop()
```



9.1 gyakorlat (egyszerű editor, 'text file'-ba iráshoz olvasáshoz) :

```
# Egyszerű szövegszerkesztő

def sansDC(ch):
    "a ch karakterláncot adja vissza az utolsó karaktere nélkül"
    uj = ""
    i, j = 0, len(ch) -1
    while i < j:
        uj = uj + ch[i]
        i = i + 1
    return uj

def fileba_ir():
    of = open(nameF, 'a')
    while 1:
        line = raw_input("Irjon be egy szovegsort (vagy <Enter>) : ")
        if line == '':
            break
        else:
            of.write(line + '\n')
    of.close()

def filebol_olvas():
    of = open(nameF, 'r')
    while 1:
        line = of.readline()
        if line == "":
            break
        # kiíratás az utolsó karakter figyelmen kívül hagyásával (= sorvége) :
        print sansDC(line)
    of.close()

nameF = raw_input('A kezelendo file neve : ')
choice = raw_input('Irjon be "i" -t irashoz, "o" -t olvasashoz : ')

if choice =='i':
    fileba_ir()
else:
    filebol_olvas()
```

9.3 gyakorlat (2 – 30 szorzótábla generálása) :

```
def tableMulti(n):
    # n-es szorzótáblát generáló függvény n (20 tag)
    # a táblát egy karakterlánc formájában adja vissza :
    i, ch = 0, ""
    while i < 20:
        i = i + 1
        ch = ch + str(i * n) + " "
    return ch

NomF = raw_input("A letrehozando file neve : ")
file = open(NomF, 'w')

# A létrehozandó táblák 2 -tõl 30 -ig :
table = 2
while table < 31:
    file.write(tableMulti(table) + '\n')
    table = table + 1
file.close()
```

9.4 gyakorlat :

```
# Ez a script azt is bemutatja, hogyan változtatjuk meg egy file tartalmát
# úgy, hogy elõször átvisszük az egészet egy listába, majd
# a listát a módosítás után kiírjuk a file-ba

def triplerEspaces(ch):
    "a fv. megháromszorozza a szavak közötti távolságot a ch stringben"
    i, uj = 0, ""
    while i < len(ch):
        if ch[i] == " ":
            uj = uj + "   "
        else:
            uj = uj + ch[i]
        i = i +1
    return uj

nameF = raw_input("File neve : ")
file = open(nameF, 'r+')              # 'r+' = mode read/write
lines = file.readlines()            # az összes sort elolvassa

n=0
while n < len(lines):
    lines[n] = triplerEspaces(lines[n])
    n =n+1

file.seek(0)                         # visszatérés a file elejére
file.writelines(lines)              # újra kiírjafile.close()
```

9.5 gyakorlat :

```
# A kezelt file egy szövegfile, aminek minden sora egy valós számot tartalmaz
# (exponens nélkül és karakterlánc formájában kódolva)

def kerErtek(ch):
    "a ch stringben megadott szam lekerekitett reprezentációja"
    f = float(ch)       # string konverziója valós számmá
    e = int(f + .5)     # átalakítás egésszé (Elõször 0.5-öt hozzáadunk
                        # a valós számhoz, hogy korrekten kerekítsünk)
    return str(e)       # visszaalakítás stringgé

fiSource = raw_input("A kezelendo file neve : ")
fiDest = raw_input("A celfile neve : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    line = fs.readline()       # a file egy sorának olvasása
    if line == "" or line == "\n":
        break
    line = kerErtek(line)
    fd.write(line +"\n")

fd.close()
fs.close()
```

9.6 gyakorlat:

```
# Két file karakterenkénti összehasonlítása :

fich1 = raw_input("Az 1. file neve : ")
fich2 = raw_input("A 2. file neve : ")
fi1 = open(fich1, 'r')
fi2 = open(fich2, 'r')

c, f = 0, 0                # karakterszámláló és flag
while 1:
    c = c + 1
    car1 = fi1.read(1)     # 1 karakter beolvasása
    car2 = fi2.read(1)     # mindegyik file-ból
    if car1 =="" or car2 ="":
        break
    if car1 != car2 :
        f = 1
        break              # eltérést talált

fi1.close()
fi2.close()

print "Ez a két file",
if f ==1:
    print "eltér a ", c, "karaktertõl"
else:
    print "azonos."
```

9.7 gyakorlat:

```
# Két szövegfile kombinálása egy új szövegfile-lá

fichA = raw_input("Az 1. file neve : ")
fichB = raw_input("A 2. file neve : ")
fichC = raw_input("A célfile neve : ")
fiA = open(fichA, 'r')
fiB = open(fichB, 'r')
fiC = open(fichC, 'w')

while 1:
    lineA = fiA.readline()
    lineB = fiB.readline()
    if lineA =="" and lineB ="":
        break              # Elértük a két file végét
    if lineA != "":
        fiC.write(lineA)
    if lineB != "":
        fiC.write(lineB)

fiA.close()
fiB.close()
fiC.close()
```

9.8 gyakorlat :

```
# Klubtagok adatainak mentése

def kodolas():
    "a beírt adatok listáját, vagy egy üres listát ad vissza"
    print "*** Írja be az adatokat (vagy <Enter> a befejezéshez) :"
    while 1:
        csaladNev = raw_input("Csaladnev : ")
        if csaladNev == "":
            return []
        utoNev = raw_input("Utonev : ")
        utcaSzam = raw_input("Cím (utca és hazszam) : ")
        irSzam = raw_input("Iranyitoszam : ")
        helyseg = raw_input("Helysegnev : ")
        tel = raw_input("Telefonszam : ")
        print csaladNev, utoNev, utcaSzam, irSzam, helyseg, tel
        ver = raw_input("<Enter> ha korrekt, <n> ha nem ")
        if ver == "":
            break
    return [csaladNev, utoNev, utcaSzam, irSzam, helyseg, tel]

def fileba_iras(lista):
    "a lista adatainak kiírása <#>-nel elválasztva a listaelemeket"
    i = 0
    while i < len(lista):
        of.write(lista[i] + "#")
        i = i + 1
    of.write("\n")              # sorvége karakter

nameF = raw_input('Célfile neve : ')
of = open(nameF, 'a')
while 1:
    tt = kodolas()
    if tt == []:
        break
    fileba_iras(tt)

of.close()
```

9.9 gyakorlat :

```
# A klub file-jának kiegészítése információval

def forditas(ch):
    "a forrásfile egy sorának átalakítása adatlistává"
    dn = ""                 # ideiglenes string az adatok kiszedéséhez
    tt = []                 # a létrehozandó lista
    i = 0
    while i < len(ch):
        if ch[i] == "#":
            tt.append(dn)   # hozzáadjuk az adatot a listához és
            dn =""          # reinicializáljuk az ideiglenes stringet
        else:
            dn = dn + ch[i]
        i = i + 1
    return tt

def kodolas(tt):
    "a tt listát adja vissza, kiegészítve a születési dátummal és a nemmel"
    print "*** Írja be az adatokat (vagy <Enter> a befejezéshez) :"
    # A listában már meglévõ adatok kiíratása :
    i = 0
    while i < len(tt):
        print tt[i],
        i = i +1
    print
    while 1:
        daNai = raw_input("Születési dátum : ")
        sexe = raw_input("Nem (f vagy n) : ")
        print daNai, sexe
        ver = raw_input("<Enter> ha korrekt, <n> ha nem ")
        if ver == "":
            break
    tt.append(daNai)
    tt.append(sexe)
    return tt

def fileba_iras(tt):
    "a tt lista adatainak kiírása <#>-nel elválasztva a listaelemeket"
    i = 0
    while i < len(tt):
        fd.write(tt[i] + "#")
        i = i + 1
    fd.write("\n")          # sorvége karakter

fSource = raw_input('Forrasfile neve : ')
fDest = raw_input('Celfile neve : ')
fs = open(fSource, 'r')
fd = open(fDest, 'w')
while 1:
    line = fs.readline()           # a forrásfile egy sorát elolvassuk
    if line =="" or line =="\n":
        break
    liste = forditas(line)         # átalakítjuk egy listává
    liste = kodolas(liste)         # kiegészítõ adatokat fûzünk hozzá
    fileba_iras(liste)             # elmentjük a célfile-ba.

fd.close()
fs.close()
```

9.10 gyakorlat :

```
# Megadott sor keresése egy szövegfile-ban :

def searchCP(ch):
    "ch -ban keresi a postai irányító számot (CP) tartalmazó stringrészt"
    i, f, ns = 0, 0, 0          # ns számlálja a # kódokat
    cc = ""                     # a létrehozandó string
    while i < len(ch):
        if ch[i] =="#":
            ns = ns +1
            if ns ==3:          # az irányítószám a 3. # után található
                f = 1           # flag
            elif ns ==4:        # nincs értelme a 4. # kód után olvasni
                break
        elif f ==1:             # az olvasott karakter részét képezi
            cc = cc + ch[i]     # a keresett CP -nek -> elmentjük
        i = i +1
    return cc

nevF = raw_input("A kezelendo file neve : ")
codeP = raw_input("A keresendo iranyitoszam : ")
fi = open(nevF, 'r')
while 1:
    line = fi.readline()
    if line =="":
        break
    if searchCP(line) == codeP:
        print line
fi.close()
```

10.2 gyakorlat (string részekre darabolása) :

```
def darabol(ch, n):
    "egy ch string n karakterbõl álló részekre darabolása"
    d, f = 0, n             # a részstring elejének és végének indexe
    tt = []                 # a létrehozandó string
    while d < len(ch):
        if f > len(ch):     # a végén túl nem tudunk darabolni
            f = len(ch)
        fr = ch[d:f]        # egy fragmentum kivágása
        tt.append(fr)       # a fragmentum hozzáadása a listához
        d, f = f, f +n      # a következõ indexek
    return tt

def invertal(tt):
    "a tt lista elemeit fordított sorrendben állítja össze"
    ch = ""                 # a létrehozandó string
    i = len(tt)             # a lista végével kezdjük
    while i > 0 :
        i = i - 1           # az utolsó elem indexe n -1
        ch = ch + tt[i]
    return ch

# Teszt :
if __name__ == '__main__':
    ch ="abcdefghijklmnopqrstuvwxyz123456789"
    lista = darabol(ch, 5)
    print lista
    print invertal(lista)
```

10.3 gyakorlat:

```
# Adott karakter indexének megkeresése egy stringben

def keres(ch, car, deb=0):
    "megkeresi a car karakter indexét a ch karakterláncban"
    i = deb
    while i < len(ch):
        if ch[i] == car:
            return i            # megtalálta a karaktert -> vége
        i = i + 1
     return -1                      # az egész karakterláncot végignézte, nincs
eredményz

# Teszt :
if __name__ == '__main__':
    print keres("Coucou c'est moi", "z")
    print keres("Juliette & Roméo", "&")
    print keres("César & Cléopâtre", "r", 5)
```

10.6 gyakorlat:

```
prefixes, suffixe = "JKLMNOP", "ack"

for p in prefixes:
    print p + suffixe
```

10.7 gyakorlat:

```
def szoSzamlalo(ch):
    "megszámolja a szavakat a ch karakterláncban"
    if len(ch) ==0:
        return 0
    nm = 1                  # a karakterlánc legalább egy szót tartalmaz
    for c in ch:
        if c == " ":        # elég a betûközöket megszámolni
            nm = nm + 1
    return nm

# Teszt :
if __name__ == '__main__':
    print szoSzamlalo("Les petits ruisseaux font les grandes rivières")
```

10.8 gyakorlat:

```
def nagybetu(car):
    "<igaz> a visszatérési értéke, ha a car nagybetû"
    if car in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
        return 1
    else:
        return 0

# Teszt :
if __name__ == '__main__':
    print nagybetu('d'), nagybetu('F')
```

10.10 gyakorlat :

```
def szoLista(ch):
    "a ch karakterláncot átalakítja szavakból álló listává"
    lista, ct = [], ""          # ct átmeneti string
    for c in ch:
        if c == " ":
            lista.append(ct)    # a listához adjuk a ch átmeneti stringet
            ct = ""             # a ch átmeneti string reinicializálása
        else:
            ct = ct + c
    if ct != "":
        lista.append(ct)        # ne felejtsük ki az utolsó szót
    return lista

# Teszt :
if __name__ == '__main__':
    print szoLista("Une hirondelle ne fait pas le printemps")
    print szoLista("")
```

10.11 gyakorlat (az előző gyakorlatokban definiált 2 függvényt használ) :

```
from exercice_10_08 import nagybetu
from exercice_10_10 import szoLista

txt = "Le nom de ce Monsieur est Alphonse"
lst = szoLista(txt)             # a mondatot átalakítja szavak listájává
for szo in lst:                 # a listában minden egyes szót megvizsgál
    if nagybetu(szo[0]):        # megvizsgálja a szó elsõ karakterét
        print szo
```

10.12 gyakorlat :

```
# karakterláncokra vonatkozó függvények modulja

def nagybetu(car):
    "visszatérési értéke <igaz> ha a car nagybetu"
    if car >= "A" and car <= "Z":
        return 1
    else:
        return 0

def kisbetu(car):
    "visszatérési értéke <igaz> ha a car kisbetu"
    if car >= "a" and car <= "z":
        return 1
    else:
        return 0

def alphab(car):
    "visszatérési értéke <igaz> ha a car az abc egy betûje"
    if nagybetu(car) or kisbetu(car):
        return 1
    else:
        return 0

# Teszt :
if __name__ == '__main__':
        print nagybetu('d'), nagybetu('F')
        print kisbetu('d'), kisbetu('F')
        print alphab('q'), alphab('P'), alphab('5')
```

10.15 gyakorlat (az előző gyakorlatokban definiált 2 függvényt használ) :

```
from exercice_10_12 import nagybetu
from exercice_10_10 import szoLista

def nagybetuSzamlalo(ch):
    "a ch stringben nagybetûvel kezdõdõ szavakat számolja meg"
    c = 0
    lst = szoLista(ch)          # mondat átalakítása szavak listájává
    for szo in lst:             # a lista mindegyik szavát elemzi
        if nagybetu(szo[0]):    # a szó elsõ betûjét ellenõrzi
            c = c +1
    return c

# Teszt :
if __name__ == '__main__':
    phrase = "Les filles Tidgout se nomment Justine et Corinne"
      print "Ez a mondat", nagybetuSzamlalo(phrase), "nagybetûvel kezdõdõ szót
tartalmaz."
```

10.16 gyakorlat (ASCII kódok táblázata) :

```
# ASCII kódtábázat

c = 32      # Elsõ <nyomtatható> ASCII

while c < 128 :                 # csak az ékezet nélküli karakterek
    print "Kód", c, ":", chr(c), "  "
    c = c + 1
```

10.18 gyakorlat (nagybetű -> és kisbetű -> nagybetű átalakítás) :

```
def csereKisNagy(ch):
    "a ch karakterláncban felcseréli egymással a kis- és a nagybetûket"
    ujC = ""                    # a létrehozandó karakterlánc
    for car in ch:
        code = ord(car)
        if car >= "A" and car <= "Z":
            code = code + 32
        elif car >= "a" and car <= "z":
            code = code - 32
        ujC = ujC + chr(code)
    return ujC

# Teszt :
if __name__ == '__main__':
    print csereKisNagy("Ferdinand-Charles de CAMARET")
```

10.20 gyakorlat (Magánhangzó számolás) :

```
def mh(car):
    "ellenõrzi, hogy car magánhangzó e"
    if car in "AEIOUYaeiouy":
        return 1
    else:
        return 0

def szamlaloMh(ch):
    "megszámolja a magánhangzókat a ch karakterláncban"
    n = 0
    for c in ch:
        if mh(c):
            n = n + 1
    return n

# Test :
if __name__ == '__main__':
    print szamlaloMh("Monty Python Flying Circus")
```

10.22 gyakorlat :

```
# Szavak számlálása egy szövegben

fiSource = raw_input("A file neve : ")
fs = open(fiSource, 'r')

n = 0           # számláló
while 1:
    ch = fs.readline()
    if ch == "":
        break
    # a beolvasott karakterlánc átalakítása szavak listájává :
    li = ch.split()
    # a szavak összegzése :
    n = n + len(li)
fs.close()
print "Ez a szövegfile összesen %s szót tartalmaz" % (n)
```

10.23 gyakorlat:

```
# Mindegyik sor elsõ karakterét nagybetûvé alakítja

fiSource = raw_input("A kezelendo file neve : ")
fiDest = raw_input("A celfile neve : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ch = fs.readline()
    if ch == "":
        break
    if ch[0] >= "A" and ch[0] <= "Z":
        # az elsõ karakter nagybetû. Nem csinálunk semmit sem.
        pass
    else:
        # A karakterlánc visszaállítása:
        pc = ch[0].upper()      # Az elsõ átalakított karakter
        rc = ch[1:]             # a karakterlánc többi része
        ch = pc + rc            # egyesítés
        # egy beépített metódust alkalmazó változat :
        # ch = ch.capitalize()
    # Átírás :
    fd.write(ch)

fd.close()
fs.close()
```

10.24 gyakorlat:

```
# Sorok egyesítése mondatokká

fiSource = raw_input("A kezelendo file neve : ")
fiDest = raw_input("A celfile neve : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')


# Elõször az elsõ sort olvassuk el :
ch1 = fs.readline()
# utána a következõ sorokat olvassuk el és egyesítjük õket, ha szükséges :
while 1:
    ch2 = fs.readline()
    if ch2 == "":
        break
    # Ha a beolvasott string nagybetûvel kezdõdik, akkor az elõzõt átírjuk a
    # célfileba, és azzal a karakterlánccal helyettesítjük, amit beolvastunk :
    if ch2[0] >= "A" and ch2[0] <= "Z":
        fd.write(ch1)
        ch1 = ch2
    # ha nem, az elõzõvel egyesítjük :
    else:
        ch1 = ch1[:-1] + " " + ch2
        # (ügyeljünk rá, hogy távolítsuk el ch1 -rõl a sorvége karaktert)

fd.write(ch1)        # ne felejtsük el átírni az utolsót !
fd.close()
fs.close()
```

10.25 gyakorlat (gömb jellemzői) :

```
# A kiindulási file egy <szövegfile>, aminek minden sora egy valós számot tartalmaz
# (string formában van kódolva)

from math import pi

def caractSphere(d):
    "megadja egy d átmérõjû gömb jellemzõit"
    d = float(d)        # az argumentum (=string) átalakítása valós számmá
    r = d/2             # sugár
    ss = pi*r**2        # fõkör területe
    se = 4*pi*r**2      # felszín
    v = 4./3*pi*r**3    # térfogat (! az elsõ osztásnak valósnak kell lenni !)
    # A lentebb alkalmazott %8.2f konverziós marker hatására a kiírt szám
    # összesen 8 helyiértéket foglal el; úgy van kerekítve, hogy
    # a tizedes pont után két helyiérték van :
    ch = "Diam. %6.2f cm Section = %8.2f cm² " % (d, ss)
    ch = ch +"Surf. = %8.2f cm². Vol. = %9.2f cm³" % (se, v)
    return ch

fiSource = raw_input("A kezelendo file neve : ")
fiDest = raw_input("A celfile neve : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    diam = fs.readline()
    if diam == "" or diam == "\n":
        break
    fd.write(caractSphere(diam) + "\n")         # kiírás fileba
fd.close()
fs.close()
```

10.26 gyakorlat :

```
# Numerikus adatok formázása
# Egy szövegfile-t kezelünk, aminek minden sora egy valós számot tartalmaz
# (exponens nélküli és string formában van kódolva)

def kerekit(real):
    ".0 -ra vagy .5 -re kerekített egész szám reprezentációja"
    ent = int(real)             # a szám egészrésze
    fra = real - ent            # törtrész
    if fra < .25 :
        fra = 0
    elif fra < .75 :
        fra = .5
    else:
        fra = 1
    return ent + fra

fiSource = raw_input("A kezelendo file neve : ")
fiDest = raw_input("A celfile neve : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    line = fs.readline()
    if line == "" or line == "\n":
        break
    n = kerekit(float(line))      # átalakítás <float> -tá, majd kerekítés
```

```
    fd.write(str(n) + "\n")         # kiírás fileba
fd.close()
fs.close()
```

10.29 gyakorlat:

```
# Szorzótáblák kiíratása

nt = [2, 3, 5, 7, 9, 11, 13, 17, 19]

def tableMulti(m, n):
     "az m-es szorzótábla n tagját adja vissza"
     ch =""
     for i in range(n):
          v = m * (i+1)              # az egyik tag kiszámolása
          ch = ch + "%4d" % (v)      # formázás 4 karakterre
     return ch

for a in nt:
     print tableMulti(a, 15)         # csak az elsõ 15 tagot
```

10.30 gyakorlat (lista bejárása):

```
lst = ['Jean-Michel', 'Marc', 'Vanessa', 'Anne',
       'Maximilien', 'Alexandre-Beno ', 'Louise']

for e in lst:
     print "%s : %s karakter" % (e, len(e))
```

10.31 gyakorlat :

```
# Az azonos adatok kiküszöbölése

lst = [9, 12, 40, 5, 12, 3, 27, 5, 9, 3, 8, 22, 40, 3, 2, 4, 6, 25]
lst2 = []

for el in lst:
     if el not in lst2:
          lst2.append(el)

lst2.sort()
print lst2
```

10.33 gyakorlat (az év összes napjának kiíratása) :

```
## Ez a változat egymásba ágyazott listákat alkalmaz ##
## (amit könnyen helyettesíthetnénk két különbözõ listával)

# Az alábbi lista két elemet tartalmaz, amik maguk is listák.
# a 0. elem a hónapok napjainak számát tartalmazza, míg
# az 1. elem a 12 hónap nevét tartalmazza :
honap = [[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
        ['Január', 'Február', 'Március', 'Április', 'Május',
         'Június', 'Július', 'Augusztus', 'Szeptember', 'Október',
         'November', 'December']]

nap = ['Vasárnap','Hétfõ','Kedd','Szerda','Csütörtök','Péntek','Szombat']

evn, hon, hen, m = 0, 0, 0, 0

while evn <365:
    evn, hon = evn +1, hon +1    # en = az év napja,  hon = a hónap napja
    hen = (evn +3) % 7           # hen js = a hét napja + offset
                                 # a kezdõnap kiválasztását teszi lehetõvé

    if hon > honap[0][m]:              # a 0. lista m-edik eleme
        hon, m = 1, m+1

    print honap[1][m], hon, nap[hen]    # az 1. lista m-edik eleme
```

10.40 gyakorlat :

```
# Erasztotenészi szita az 1 és 999 közé esõ prímszámok megkereséséhez

# Egy 1-esekbõl álló 1000 elemû lista létrehozása (az indexei 0-tól 999-ig
mennek) :
lst = [1]*1000
# A lista bejárása a 2 indexértéktõl kezdve:
for i in range(2,1000):
    # a listának azokat az elemeit, melyek indexe
    # az i-nek többszörösei nullává tesszük :
    for j in range(i*2, 1000, i):
        lst[j] = 0

# Kiíratjuk azoknak az elemeknek az indexét, melyek értéke 1 maradt
# (a 0 elemet nem vesszük figyelembe) :
for i in range(1,1000):
    if lst[i]:
        print i,
```

10.43 gyakorlat (Véletlenszám generátor tesztje) :

```
# A véletlenszám generátor tesztelése

from random import random     # 0 és 1 közé esõ valós véletlenszámot sorsol

n = raw_input("Hany veletlenszamot sorsoljon (default = 1000) : ")
if n == "":
    nVal =1000
else:
    nVal = int(n)

n = raw_input("A reszintervallumok szama 0-1 -ben (2 es "
              + str(nVal/10) + " között, default =5) : ")
if n == "":
    nFra =5
else:
    nFra = int(n)

if nFra < 2:
    nFra =2
elif nFra > nVal/10:
    nFra = nVal/10

print nVal, "húzás értékeinek megoszlása ..."
listVal = [0]*nVal                    # nullákból álló listát hoz létre
for i in range(nVal):                 # majd mindegyik elemet módosítja
    listVal[i] = random()

print "Megszámolja az értékeket a(z)", nFra, "részinterv. mindegyikében ..."
listCompt = [0]*nFra                  # létrehozza a számlálók listáját
# az értékek listájának bejárása :
for valeur in listVal:
    # megkeresi az értéket tartalmazó részintervallum indexét :
    index = int(valeur*nFra)
    # incrementálja a megfelelõ számlálót :
    listCompt[index] = listCompt[index] +1

# kiíratja a számlálók állapotát :
for compt in listCompt:
    print compt,
```

10.44 gyakorlat : kártyahúzás

```
# Kártyahúzás
from random import randrange

colours = ['Pique', 'Trefle', 'Carreau', 'Coeur']
values = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'junga', 'dáma', 'király', 'ász']

# Az 52 kártya listájának a létrehozása :
card =[]
for coul in colours:
     for val in values:
          card.append("%s %s" % (coul, str(val)))

# V etlenszer? hz  :
while 1:
     k = raw_input("<c> kártyát húz, <Enter> vége")
     if k =="":
          break
     r = randrange(52)
     print card[r]
```

10.45 gyakorlat : Szótár létrehozása és megtekintése

```
def megnez():
    while 1:
        nev = raw_input("Irjon be egy nevet (vagy <enter> a befejezeshez) : ")
        if nev == "":
            break
        if dico.has_key(nev):           # a név a listában van ? ?
            item = dico[nev]            # megnézzük
            eletkor, magassag = item[0], item[1]
            print "Nev : %s - eves : %s ans - magassag : %s m."\
                  % (nev, eletkor, magassag)
        else:
            print "*** ismeretlen nev ! ***"

def kitolt():
    while 1:
        nev = raw_input("Irjon be egy nevet (vagy <enter> a befejezéshez) : ")
        if nev == "":
            break
        eletkor = int(raw_input("Irja be az eletkort (egeszszam !) : "))
        magassag = float(raw_input("Irja be a magassagot (meterben) : "))
        dico[nev] = (eletkor, magassag)

dico ={}
while 1:
    choice = raw_input("Valasszon : (K)itolt - (M)egnez - (V)ege : ")
    if choice.upper() == 'V':
        break
    elif choice.upper() == 'K':
        kitolt()
    elif choice.upper() == 'M':
        megnez()
```

10.46 gyakorlat : a kulcsok és értékek felcserélése egy szótárban

```
# A kulcsok és értékek felcserélése egy szótárban

def inverse(dico):
    "egy új szótár létrehozása lépésről lépésre"
    dic_inv ={}
    for key in dico:
        item = dico[key]
        dic_inv[item] = key

    return dic_inv

# programteszt :

dico = {'Computer':'Szamitogep',
        'Mouse':'Eger',
        'Keyboard':'Billentyuzet',
        'Hard disk':'Merev lemez',
        'Screen':'Kepernyo'}

print dico
print inverse(dico)
```

10.47 gyakorlat : hisztogram

```
# Egy szövegben elõforduló betûk elõfordulási gyakoriságának hisztogramja

nFile = raw_input('Filenev : ')
fi = open(nFile, 'r')
text = fi.read()                # a file átalakítása karakterlánccá
fi.close()

print text
dico ={}
for c in text:
    c = c.upper()               # minden betût nagybetûvé alakít
    dico[c] = dico.get(c, 0) +1

list = dico.items()
list.sort()
print list
```

10.48 gyakorlat :

```
# Szöveg szavainak előfordulási gyakoriságából készített hisztogram

nFile = raw_input('Kezelendo file neve : ')
fi = open(nFile, 'r')
text = fi.read()
fi.close()

# annak erdekeben, hogy a szoeg szavai konnyen elklulonithetok legyenek, a nem
# betu karaktereket betukozze alakitjuk  :

alpha = "abcdefghijklmnopqrstuvwxyzíéáűúőóüöÍÉÁŰÚŐÓÜÖ"

letters = ''            # a lerehozand uj karakterlanc
for c in text:
    c = c.lower()       # minden betut kisbetuve alakit
    if c in alpha:
```

```
        letters = letters + c
    else:
        letters = letters + ' '

# az eredménystring átalakítása szavak listájává :
words = letters.split()

# a hisztogram elkészítése :
dico ={}
for m in words:
    dico[m] = dico.get(m, 0) +1

liste = dico.items()

# az eredménylista rendezése :
liste.sort()

# kiiratas :
for item in liste:
    print item[0], ":", item[1]
```

10.49 gyakorlat:

```
# szoveg kodolasa szotarba

nFile = raw_input('Kezelendo file neve : ')
fi = open(nFile, 'r')
text = fi.read()
fi.close()

# Ugy tekintjük, hogy a szavak olyan karakterek sorozatai, amik az alabbi
# stringnek kepezik reszeit. Minden mas karakter szeparator :

alpha = "abcdefghijklmnopqrstuvwxyzáéíóőúű"

# szotar letrehozasa :
dico ={}
# a szoveg oszes karakterenek bejarasa :
i =0                    # az eppen beolvasott karakter indexe
word =""                # munkavaltozo : az eppen beolvasott szo
for c in text:
    c = c.lower()       # minden betut kisbetuve alakit

    if c in alpha:      # alfabetikus karakter => egy szo belsejeben vagyunk
        word = word + c
    else:               # nem alfabetikus karakter => szo vege
        if word !="":   # hogy figyelmen kivül hagyjuk az egymast koveto nem
                        # alfabetikus karaktereket minden szonak,
                        #lerehozunk egy indexlistat:

            if dico.has_key(word):       # már listába vett szo :
                dico[word].append(i)     # egy index hozzaadasa a listahoz
            else:                        # eloszor elofordulo szo :
                dico[word] =[i]          # indexek listájanak letrehozasa
            word =""     # a kovetkezo szo olvasasanak elokeszitese
    i = i + 1            # a kovetkezo karakter indexe

# A szotar kiiratasa :
for key, value in dico.items():
    print key, ":", value
```

10.50 gyakorlat : Szótár mentése (10.45 gyakorlat kiegészítése).

```
def record():
    file = raw_input("Irja be az elmentendo file nevet : ")
    ofi = open(file, "w")
    # az elõzõleg listává alakított szótár bejárása :
    for key, value in dico.items():
        # stringformázás alkalmazása :
        ofi.write("%s@%s#%s\n" % (key, value[0], value[1]))
    ofi.close()

def readFile():
    file = raw_input("Irja be az elmentett file nevet : ")
    try:
        ofi = open(file, "r")
    except:
        print "*** nem létezõ file ***"
        return

    while 1:
        line = ofi.readline()
        if line =='':              # filevége detektálása
            break
        enreg = line.split("@")    # [key,value] lista visszaállítása
        key = enreg[0]
        value = enreg[1][:-1]      # sorvége karakter kiküszöbölése
        data = value.split("#")    # [age, height] lista visszaállítása
        age, height = int(data[0]), float(data[1])
        dico[key] = (age, height)   # szótár rekonstruálása
    ofi.close()
```

Ezt a két függvényt a főprogram elején, illetve végén hívhatjuk,mint az alábbi példában :

```
dico ={}
readFile()
while 1:
    choice = raw_input("Valasszon : (K)itolt - (M)egnez - (V)ege : ")
    if choice.upper() == 'V':
        break
    elif choice.upper() == 'K':
        fill()
    elif choice.upper() == 'M':
        consult()
record()
```

10.51 gyakorlat : Programvégrehajtás vezérlése szótárral

Ez a gyakorlat az előzőt egészíti ki. Hozzáadunk még két függvényt és átírjuk a főprogramot, hogy egy szótárral vezéreljük a programvégrehajtást :

```
def finish():
    print "*** Munka vége ***"
    return 1                        # a ciklus befejezésének előidézése

def other():
    print "Kérem válasszon V, H, B, M vagy K."
```

```
dico ={}
func ={"V":readFile, "H":fill, "B":consult,
       "M":writeFile, "K":finish}
while 1:
    choix = raw_input("Válasszon :\n" +\
    "(V)isszaolvasunk egy mar letezo mentett szotarat egy filebol\n" +\
    "(H)hozzaadunk az aktualis szotarhoz ujabb adatokat\n" +\
    "(B)elenezunk az aktualis szotarba\n" +\
    "(M)enti az aktualis szotarat egy fileba\n" +\
    "(K)ilep : ")
    # az alábbi utasítás minden választásra más függvényt hív
    # a <func> szótár segítségével :
    if func.get(choice, other)():
        break
    # Rem : minden itt hívott függvény alapértelmezett visszatérési értéke ,
    # <None> kivéve a finish() függvényt, esetében 1 => kilépés a ciklusból
```

12.1 gyakorlat :

```
class Domino:
    def __init__(self, pa, pb):
        self.pa, self.pb = pa, pb

    def kiir_pontok(self):
        print "A oldal :", self.pa,
        print "B oldal :", self.pb

    def ertek(self):
        return self.pa + self.pb

# Tesztprogram :

d1 = Domino(2,6)
d2 = Domino(4,3)

d1.kiir_pontok()
d2.kiir_pontok()

print "Összes pont :", d1.ertek() + d2.ertek()

lista_dominok = []
for i in range(7):
    lista_dominok.append(Domino(6, i))

vt =0
for i in range(7):
    lista_dominok[i].kiir_pontok()
    vt = vt + lista_dominok[i].ertek()

print "Összpontszám :", vt
```

12.3 gyakorlat:

```
class auto:
    def __init__(self, marka = 'Ford', szin = 'piros'):
        self.szin = szin
        self.marka = marka
        self.vezeto = 'senki'
        self.sebesseg = 0

    def gyorsit(self, gyorsulas, idotartam):
        if self.vezeto =='senki':
            print "Ennek az autónak nincs vezetője !"
        else:
            self.sebesseg = self.sebesseg + gyorsulas * idotartam

    def valaszt_sofort(self, nev):
        self.vezeto = nev

    def kiir_mindent(self):
        print "%s %s vezeti %s, sebesseg = %s m/s" % \
            ( self.szin, self.marka, self.vezeto, self.sebesseg)

a1 = auto('Peugeot', 'kék')
a2 = auto(szin = 'zöld')
a3 = auto('Mercedes')
a1.valaszt_sofort('Romeo')
a2.valaszt_sofort('Juliette')
a2.gyorsit(1.8, 12)
a3.gyorsit(1.9, 11)
a1.kiir_mindent()
a2.kiir_mindent()
a3.kiir_mindent()
```

12.4 gyakorlat:

```
class Muhold:
    def __init__(self, nev, tomeg =100, sebesseg =0):
        self.nev, self.tomeg, self.sebesseg = nev, tomeg, sebesseg

    def lokes(self, ero, idotartam):
        self.sebesseg = self.sebesseg + ero * idotartam / self.tomeg

    def energia(self):
        return self.tomeg * self.sebesseg**2 / 2

    def kiir_sebesseg(self):
        print "%s Műhold sebessége = %s m/s" \
                          % (self.nev, self.sebesseg)

# Tesztprogram :

s1 = Muhold('Zoé', tomeg =250, sebesseg =10)

s1.lokes(500, 15)
s1.kiir_sebesseg()
print s1.energia()
s1.lokes(500, 15)
s1.kiir_sebesseg()
print s1.energia()
```

12.5-12.6 gyakorlatok (henger- és kúp-osztály) :

```
# Leszármaztatott osztályok - polimorfizmus

class Kor:
    def __init__(self, sugar):
        self.sugar = sugar

    def terulet(self):
        return 3.1416 * self.sugar**2

class Henger(Kor):
    def __init__(self, sugar, magassag):
        Kor.__init__(self, sugar)
        self.magassag = magassag

    def terfogat(self):
        return self.terulet()*self.magassag

        # a terulet() metódust a szülőosztálytól örökli

class Kup(Henger):
    def __init__(self, sugar, magassag):
        Henger.__init__(self, sugar, magassag)

    def terfogat(self):
        return Henger.terfogat(self)/3

        # ez az új terfogat() metódus helyettesíti a
        # zülőosztálytól örökölt metódust (polimorfizmus példa)

#  Tesztprogram :

henger = Henger(5, 7)
print "Henger alapterülete =", henger.terulet()
print "Hengertérfogat =", henger.terfogat()

kup = Kup(5,7)
print "Kúp alapterülete =", kup.terulet()
print "Kúptéfogat =", kup.terfogat()
```

12.7 gyakorlat :

```
# Kártyahúzáa

from random import randrange

class KartyaJatek:
    """Kártyajáték"""
    # osztályattribútumok (minden példány esetében közösek) :
    couleur = ('Pique', 'Trefle', 'Carreau', 'Coeur')
    ertek = (0, 0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'junga', 'dama', 'kiraly', 'asz')

    def __init__(self):
        "Az 52 kártya listájának létrehozása"
        self.kartya =[]
        for coul in range(4):
            for val in range(13):
                self.kartya.append((val +2, coul))   # az érték 2-vel kezdõdik

    def kartya_neve(self, c):
        "A c kártya értékét adja meg, en clair"
        return "%s %s" % (self.couleur[c[1]], self.ertek[c[0]])

    def kever(self):
        "Megkeveri a kártyákat"
        t = len(self.kartya)             # a maradék kártyák száma
        # a keveréshez azonos számú cserét végzünk :
        for i in range(t):
            # 2 listabeli hely véletlenszerû sorsolása :
            h1, h2 = randrange(t), randrange(t)
            # felcseréljük az ezeken a helyeken lévõ kártyákat :
            self.kartya[h1], self.kartya[h2] = self.kartya[h2], self.kartya[h1]

    def huz(self):
        "A pakli elsõ kártyájának kihúzása"
        t = len(self.kartya)             # ellenõrizzük, hogy maradt-e kártya
        if t >0:
            kartya = self.kartya[0]       # kiválasztjuk az elsõ kártyát
            del(self.kartya[0])          # kivonjuk a játékból
                return kartya                    # megadjuk a másolatát a hívó
programnak
        else:
            return None                 # fakultativ

### Tesztprogramme  :

if __name__ == '__main__':
    jatek = KartyaJatek()                # objektum létrehozása
    jatek.kever()                        # kártyák keverése
    for n in range(53):                  # 52 kártya húzása :
        c = jatek.huz()
        if c == None:                    # egyetlen kártya sem marad
            print 'Vege !'           # a listában
        else:
            print jatek.kartya_neve(c)      # a kártya értéke és színe
```

12.8 gyakorlat :

```
(Feltételezem, hogy az előző gyakorlatot exercice_12_07.py néven mentettük el.)
# Kartyacsata

from exercice_12_07 import KartyaJatek

jatekA = KartyaJatek()        # az elso jatek letrehozasa
jatekB = KartyaJatek()        # a masodik jatek letrehozasa
jatekA.kever()               # mange de chacun
jatekB.kever()
pA, pB = 0, 0               # compteurs de points des joueurs A et B

# mindegyik jatek eseteben 52-szer huzunk :
for n in range(52):
    cA, cB = jatekA.huz(), jatekB.huz()
    vA, vB = cA[0], cB[0]   # a kartyak ertekei
    if vA > vB:
        pA += 1
    elif vB > vA:
        pB += 1             # (semmi sem tortenik, ha vA = vB)
    # a kartyak es a pontok egymasutani kiiratasa :
    print "%s * %s ==> %s * %s" % (jatekA.kartya_neve(cA),
                                   jatekB.kartya_neve(cB), pA, pB)

print "Az A jatekos %s pontot nyert, a B jatekos %s pontot nyert." % (pA, pB)
```

13.6 gyakorlat :

```
from Tkinter import *

def circle(can, x, y, r, colour ='white'):
    "<r> sugaru kor rajzolasa a vaszon <can> <x,y> pontjaba"
    can.create_oval(x-r, y-r, x+r, y+r, fill =colour)

class Application(Tk):
    def __init__(self):
        Tk.__init__(self)        # a szuloosztaly constructora
        self.can =Canvas(self, width =475, height =130, bg ="white")
        self.can.pack(side =TOP, padx =5, pady =5)
        Button(self, text ="Vonat", command =self.drawing ).pack(side =LEFT)
        Button(self, text ="Hello", command =self.kukucs).pack(side =LEFT)
        Button(self, text ="Világít34", command =self.light34).pack(side =LEFT)

    def drawing (self):
        "4 vagon letrehozasa a vasznon"
        self.w1 = Wagon(self.can, 10, 30)
        self.w2 = Wagon(self.can, 130, 30, 'dark green')
        self.w3 = Wagon(self.can, 250, 30, 'maroon')
        self.w4 = Wagon(self.can, 370, 30, 'purple')

    def kukucs(self):
        "emberek jelennek meg bizonyos ablakokban"
        self.w1.perso(3)        # 1. vagon, 3. ablak
        self.w3.perso(1)        # 3. vagon, 1. ablak
        self.w3.perso(2)        # 3. vagon, 2. ablak
        self.w4.perso(1)        # 4. vagon, 1. ablak

    def light34(self):
        "a vilagitas bekapcsolasa a 3 & 4 vagonban"
```

```
        self.w3.light()
        self.w4.light()

class Wagon:
    def __init__(self, canvas_, x, y, colour ='navy'):
        "egy kis vagon rajza a <canvas_> vásznon <x,y> -ban"
        # paraméterek tárolása példány-változókban :
        self.canvas_, self.x, self.y = canvas_, x, y
        # alap téglalap : 95x60 pixel :
        canvas_.create_rectangle(x, y, x+95, y+60, fill =colour)
        # 3 ablak 25x40 pixeles, 5 pixel távolságra :
        self.wind =[]    # az ablakok hivatkozasainak tarolasara
        for xf in range(x +5, x +90, 30):
            self.wind.append(canvas_.create_rectangle(xf, y+5,
                                xf+25, y+40, fill ='black'))
        # két 12 pixel sugarú kerék  :
        circle(canvas_, x+18, y+73, 12, 'gray')
        circle(canvas_, x+77, y+73, 12, 'gray')

    def perso(self, wind):
        "egy ember jelenik meg a <wind> ablakban"
        # az ablakok koordinatainak kiszamolasa :
        xf = self.x + wind*30 -12
        yf = self.y + 25
        circle(self.canvas_, xf, yf, 10, "pink")      # arc
        circle(self.canvas_, xf-5, yf-3, 2)   # balszem
        circle(self.canvas_, xf+5, yf-3, 2)   # jobbszem
        circle(self.canvas_, xf, yf+5, 3)     # szaj

    def light(self):
        "felkapcsolja a vagon belso vilagitasat"
        for f in self.wind:
            self.canvas_.itemconfigure(f, fill ='yellow')

app = Application()
app.mainloop()
```

13.21 gyakorlat :

```
from Tkinter import *
# A modul diszken levo file-ok keresesere szolgalo
# standard dialogbox-okat allit elo :
from tkFileDialog import asksaveasfile, askopenfile

class Application(Frame):
    '''Alkalmazasablak'''
    def __init__(self):
        Frame.__init__(self)
        self.master.title("Színszótár létrehozása")

        self.dico ={}       # szotar letrehozasa

        # A widget-ek ket keretben (Frame) vannak csoportositva :
        frUpp =Frame(self)      # 6 widget -et tartalmazo felso frame
        Label(frUpp, text ="Szín neve :",
              width =20).grid(row =1, column =1)
        self.enName =Entry(frUpp, width =25)        # adatbeviteli mezo
        self.enName.grid(row =1, column =2)         # a szinnek
        Button(frUpp, text ="Már létezik ?", width =12,
               command =self.searchColour).grid(row =1, column =3)
        Label(frUpp, text ="Megf. hexa kód :",
              width =20).grid(row =2, column =1)
        self.enCode =Entry(frUpp, width =25)        # adatbeviteli mezo
        self.enCode.grid(row =2, column =2)         # a hexakodnak
        Button(frUpp, text ="Teszt", width =12,
               command =self.testColour).grid(row =2, column =3)
        frUpp.pack(padx =5, pady =5)

        frLow =Frame(self)      # a maradekot tartalmazo also frame
        self.test = Label(frLow, bg ="white", width =45,    # tesztzona
                          height =7, relief = SUNKEN)
        self.test.pack(pady =5)
        Button(frLow, text ="A szín hozzáadása a szótárhoz",
               command =self.addColour).pack()
        Button(frLow, text ="Szótár kiírása", width =25,
               command =self.record).pack(side = LEFT, pady =5)
        Button(frLow, text ="Szótár visszaállítása", width =25,
               command =self.restore).pack(side =RIGHT, pady =5)
        frLow.pack(padx =5, pady =5)
        self.pack()

    def addColour(self):
        "az aktualis szin hozzaadasa a szotarhoz"
        if self.testColour() ==0:        # volt definialva szin ?
            return
        name = self.enName.get()
        if len(name) >1:                 # elutasitja a tul rovid nevet
            self.dico[name] =self.cHexa
        else:
            self.test.config(text ="%s : inkorrekt nev" % name, bg ='white')

    def searchColour(self):
        "egy a szotarba mar beirt szin keresese"
        name = self.enName.get()
        if self.dico.has_key(name):
            self.test.config(bg =self.dico[name], text ="")
        else:
            self.test.config(text ="%s : ismeretlen szin" % name, bg ='white')

    def testColour(self):
```

```
        "hexakod ervenyessegenek igazolasa. - a megfelelo szin kiirasa."
        try:
            self.cHexa =self.enCode.get()
            self.test.config(bg =self.cHexa, text ="")
            return 1
        except:
            self.test.config(text ="Inkorrekt szin kodolasa", bg ='white')
            return 0

    def record(self):
        "szotar kiirasa szovegfile-ba"
        # Ez a metodus egy standard dialogbox-ot alkalmaz a diszken levo file
        # kivalasztasara. A Tkinter a tkFileDialog modulban egy sor fggvyt ad,
        # amik ezekhez a # dialogbox-okhoz vannak asszocialva.
        # Az alabbi fuggveny egy irasra megnyitott file-objektumot ad vissza :
        ofi =asksaveasfile(filetypes=[("Szöveg",".txt"),("Minden","*")])
        for key, value in self.dico.items():
            ofi.write("%s %s\n" % (key, value))
        ofi.close()

    def restore(self):
        "szotar visszaallitasa tarolt filebol"
        # Az alabbi fuggveny egy olvasasra megnyitott file-objektumot
        # ad visszateresi erteknek :
        ofi =askopenfile(filetypes=[("Szöveg",".txt"),("Minden","*")])
        lignes = ofi.readlines()
        for li in lignes:
            cv = li.split()     # a megfelelo kulcs es ertek kiszedese
            self.dico[cv[0]] = cv[1]
        ofi.close()

if __name__ == '__main__':
    Application().mainloop()
```

13.22 gyakorlat (3. változat) :

```
from Tkinter import *
from random import randrange
from math import sin, cos, pi

class FaceDom:
    def __init__(self, can, val, pos, size =70):
        self.can =can
        x, y, c = pos[0], pos[1], size/2
        self. square = can.create_rectangle(x -c, y-c, x+c, y+c,
                                            fill ='ivory', width =2)
        d = size/3
        # a pontok elrendezese az oldalon, mind a 6 esetre :
        self.pDispo = [((0,0),),
                       ((-d,d),(d,-d)),
                       ((-d,-d), (0,0), (d,d)),
                       ((-d,-d),(-d,d),(d,-d),(d,d)),
                       ((-d,-d),(-d,d),(d,-d),(d,d),(0,0)),
                       ((-d,-d),(-d,d),(d,-d),(d,d),(d,0),(-d,0))]

        self.x, self.y, self.dim = x, y, size/15
        self.pList =[]      # ennek az oldalnak a pontjait tartalmazo lista
        self.draw_points(val)

    def draw_points(self, val):
        # a val értéknek megfelelő pontok rajzainak a létrehozása :
```

```
        disp = self.pDispo[val -1]
        for p in disp:
            self.circle(self.x +p[0], self.y +p[1], self.dim, 'red')
        self.val = val

    def circle(self, x, y, r, colo):
        self.pList.append(self.can.create_oval(x-r, y-r, x+r, y+r, fill=colo))

    def erase(self, flag =0):
        for p in self.pList:
            self.can.delete(p)
        if flag:
            self.can.delete(self.square)

class Project(Frame):
    def __init__(self, width_, height_):
        Frame.__init__(self)
        self.width_, self.height_ = width_, height_
        self.can = Canvas(self,bg='dark green', width =width_, height =height_)
        self.can.pack(padx =5, pady =5)
        # az elhelyezendő gombok és az eseménykezelőik listája :
        bList = [("A", self.buttonA), ("B", self.buttonB),
                 ("C", self.buttonC), ("Quitter", self.buttonQuit)]
        bList.reverse()         # a lista sorrendjét megfordítja
        for b in bList:
            Button(self, text =b[0], command =b[1]).pack(side =RIGHT, padx=3)
        self.pack()
        self.die =[]            # a kockákat tartalmazó lista
        self.actu =0            # az aktuálisan kiválasztott kocka hivatkozása

    def buttonA(self):
        if len(self.die):
            return              # mert a rajzok m  leznek !
        a, da = 0, 2*pi/13
        for i in range(13):
            cx, cy = self.width_/2, self.height_/2
            x = cx + cx*0.75*sin(a)            # kör elhelyezéséhez,
            y = cy + cy*0.75*cos(a)            # a trigonometriát alkalmazzuk !
            self.die.append(FaceDom(self.can, randrange(1,7) , (x,y), 65))
            a += da

    def buttonB(self):
        # a kiválasztott kocka pontértékét növeli. A következő kockára tér át :
        v = self.die[self.actu].val
        v = v % 6
        v += 1
        self.die[self.actu].erase()
        self.die[self.actu].draw_points(v)
        self.actu += 1
        self.actu = self.actu % 13

    def buttonC(self):
        for i in range(len(self.die)):
            self.die[i].erase(1)
        self.die =[]
        self.actu =0

    def buttonQuit(self):
        self.master.destroy()

Project(600, 600).mainloop()
```

16.1 gyakorlat ("musique" nevű adatbázis létrehozása) :

```
# Egy kis adatbázis létrehozása és feltöltése

import gadfly
import os

path_ = os.getcwd()        # aktualis konyvtar

connection = gadfly.gadfly()
connection.startup("music", path_)
cur = connection.cursor()
request = "create table zeneszerzok (eloado varchar, ev_szul integer,\
          ev_halal integer)"
cur.execute(request)
request = "create table muvek (eloado varchar, cim varchar,\
          ido integer, interpr varchar)"
cur.execute(request)

print "Bevitt adatok kiirasa a zeneszerzok adattablaba :"
while 1:
    nv = raw_input("Zeneszerzo neve (<Enter> befejezes) : ")
    if nm =='':
        break
    esz = raw_input("Szuletes eve : ")
    eha = raw_input("Halal eve : ")
    request ="insert into zeneszerzok(eloado, ev_szul, ev_halal) values \
                ('%s', %s, %s)" % (nv, esz, eha)
    cur.execute(request)
# Ellenorzeskent kiirja a bevitt adatokat :
cur.execute("select * from zeneszerzok")
print cur.pp()

print "Bevitt adatok rögzitese a muvek adattablaba :"
while 1:
    nev_ = raw_input("Zeneszerzo neve (<Enter> befejezes) : ")
    if nev_ =='':
        break
    cim_ = raw_input("Zenemu cime : ")
    ido_ = raw_input("idotartam (perc) : ")
    int = raw_input("eloado : ")
    request ="insert into muvek(eloado, cim, ido, interpr) values \
                ('%s', '%s', %s, '%s')" % (nev_, cim_, ido_, int)
    cur.execute(request)
# Ellenorzeskent kiirja a bevitt adatokat :
cur.execute("select * from muvek")
print cur.pp()

connection.commit()
```

18.2 gyakorlat :

```
#####################################
# Mozgó célpont bombázása           #
# (C) G. Swinnen - Avril 2004 - GPL #
#####################################

from Tkinter import *
from math import sin, cos, pi
from random import randrange
from threading import Thread
import time
```

```
class Canon:
    """Ágyúrajz"""
    def __init__(self, boss, num, x, y, irany):
        self.boss = boss            # a vászon hivatkozása
        self.num = num              # az ágyú száma a listában
        self.x1, self.y1 = x, y     # ágyú forgástengelye
        self.irany = irany          # lövés iránya (-1:balra, +1:jobbra)
        self.lagyucso = 30          # ágyúcsõ hossza
        # ágyúcsõ rajzolása (vízszintes) :
        self.x2, self.y2 = x + self.lagyucso * irany, y
        self.cso = boss.create_line(self.x1, self.y1,
                                    self.x2, self.y2, width =10)
        # ágyútest rajzolása (színes kör) :
        self.rc = 15                # a kör sugara
        self.test = boss.create_oval(x -self.rc, y -self.rc, x +self.rc,
                                     y +self.rc, fill ='black')
        # egy lövedék elõrajzolása (kiinduláskor egy pont) :
        self.lovedek = boss.create_oval(x, y, x, y, fill='red')
        self.anim = 0
        # a vászon szélességének és magasságának meghatározása :
        self.xMax = int(boss.cget('width'))
        self.yMax = int(boss.cget('height'))

    def iranyzas(self, angle):
        "ágyú dõlésszögének beállítása"
        # rem : az <angle> (szög) paramétert stringként kapja.
        # át kell alakítani valós számmá, majd radiánná :
        self.angle = float(angle)*2*pi/360
        self.x2 = self.x1 + self.lagyucso * cos(self.angle) * self.irany
        self.y2 = self.y1 - self.lagyucso * sin(self.angle)
        self.boss.coords(self.cso, self.x1, self.y1, self.x2, self.y2)

    def tuz(self):
        "golyó kilövésének indítása"
        # céltárgy hivatkozása :
        self.celtargy = self.boss.master.celtargy
        if self.anim ==0:
            self.anim =1
            # lövedék kezdõ pozíciója (ez az ágyú torka) :
            self.xo, self.yo = self.x2, self.y2
            v = 20              # kezdõ sebesség
            # ennek a sebességnek a függõleges és vízszintes komponensei :
            self.vy = -v *sin(self.angle)
            self.vx = v *cos(self.angle) *self.irany
            self.animacio_lovedek()

    def animacio_lovedek(self):
        "lövedék animációja (ballisztikus pálya)"
        # lövedék pozícionálása koordinátái újradefiniálásával :
        self.boss.coords(self.lovedek, self.xo -3, self.yo -3,
                                       self.xo +3, self.yo +3)
        if self.anim >0:
            # a következõ pozíció számolása :
            self.xo += self.vx
            self.yo += self.vy
            self.vy += .5
            self.test_akadaly()        # akadályba ütközött ?
            self.boss.after(1, self.animacio_lovedek)
        else:
            # animáció vége :
            self.boss.coords(self.lovedek, self.x1, self.y1, self.x1, self.y1)
```

```
    def test_akadaly(self):
        "Eldöntjük: a lövedék célba talált-e vagy elérte-e a játéktér határát"
        if self.yo >self.yMax or self.xo <0 or self.xo >self.xMax:
            self.anim =0
            return
        if self.yo > self.celtargy.y -3 and self.yo < self.celtargy.y +18 \
        and self.xo > self.celtargy.x -3 and self.xo < self.celtargy.x +43:
            # megrajzoljuk a lövedék robbanását (narancssárga kör) :
            self.robbanas = self.boss.create_oval(self.xo -10,
                        self.yo -10, self.xo +10, self.yo +10,
                        fill ='orange', width =0)
            self.boss.after(150, self.robbanas_vege)
            self.anim =0

    def robbanas_vege(self):
        "a robbanási kör törlése - a pontszám kezelése"
        self.boss.delete(self.robbanas)
        # a siker jelzése a master-ablaknak :
        self.boss.master.goal()

class VezerloPult(Frame):
    """Egy ágyúhoz asszociált vezérlõpult"""
    def __init__(self, boss, canon):
        Frame.__init__(self, bd =3, relief =GROOVE)
        self.score =0
        s =Scale(self, from_ =88, to =65,
                 troughcolor ='dark grey',
                 command =canon.iranyzas)
        s.set(45)                       # a lövés kezdõ szöge
        s.pack(side =LEFT)
        Label(self, text ='Dõlésszög').pack(side =TOP, anchor =W, pady =5)
        Button(self, text ='Tûz !', command =canon.tuz).\
                                    pack(side =BOTTOM, padx =5, pady =5)
        Label(self, text ="pont").pack()
        self.pontok =Label(self, text=' 0 ', bg ='white')
        self.pontok.pack()
        # positionner a gauche ou a droite suivant le sens du canon :
        gd =(LEFT, RIGHT)[canon.irany == -1]
        self.pack(padx =3, pady =5, side =gd)

    def pontHozzaadasa(self, p):
        "a pontszám növelése vagy csökkentése"
        self.score += p
        self.pontok.config(text = ' %s ' % self.score)

class Celtargy:
    """céltárgyként szolgáló grafikus objektum"""
    def __init__(self, can, x, y):
        self.can = can             # a vászon hivatkozása
        self.x, self.y = x, y
        self.celtargy = can.create_oval(x, y, x+40, y+15, fill ='purple')

    def elmozditas(self, dx, dy):
        "a céltárgy dx,dy elmozdítása"
        self.x += dx
        self.y += dy
        self.can.move(self.celtargy, dx, dy)
        return self.x, self.y
```

```
class Thread_celtargy(Thread):
    """a céltárgy animációját irányító thread-objektum"""
    def __init__(self, app, celtargy):
        Thread.__init__(self)
        self.celtargy = celtargy    # az elmozdítandó objektum
        self.app = app              # az alkalmazásablak hivatkozása
        self.sx, self.sy = 6, 3     # távolság- és idõnövekmények
        self.dt =.3                 # az animációhoz

    def run(self):
        "animálás, amíg az alkalmazásablak létezik"
        while self.app != None:
            x, y = self.celtargy.elmozditas(self.sx, self.sy)
            if x > self.app.xm -50 or x < self.app.xm /5:
                self.sx = -self.sx
            if y < self.app.ym /2 or y > self.app.ym -20:
                self.sy = -self.sy
            time.sleep(self.dt)

    def stop(self):
        "a thread zárása, ha az alkalmazásablak zárva van"
        self.app =None

    def gyorsit(self):
        "a mozgás gyorsítása"
        self.dt /= 1.5

class Application(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.master.title('<<< Lövés mozgó céltárgyra >>>')
        self.pack()
        self.xm, self.ym = 600, 500
        self.jatek = Canvas(self, width =self.xm, height =self.ym,
                            bg ='ivory', bd =3, relief =SUNKEN)
        self.jatek.pack(padx =4, pady =4, side =TOP)

        # Egy ágyú és egy vezérlõ pult objektum létrehozása :
        x, y = 30, self.ym -20
        self.agyu =Canon(self.jatek, 1, x, y, 1)
        self.pult =VezerloPult(self, self.agyu)

        # mozgó célpont létrehozása :
        self.celtargy = Celtargy(self.jatek, self.xm/2, self.ym -25)
        # mozgó célpont animálása saját thread-del :
        self.tc = Thread_celtargy(self, self.celtargy)
        self.tc.start()
        # az összes thread leállítása amikor zárjuk az ablakot :
        self.bind('<Destroy>',self.zaras_threadek)

    def goal(self):
        "a célpontot eltaláltuk"
        self.pult.pontHozzaadasa(1)
        self.tc.gyorsit()

    def zaras_threadek(self, evt):
        "a célpont animációs thread-jének leállítása"
        self.tc.stop()
if __name__ =='__main__':
    Application().mainloop()
```

1.10 A« How to think like a computer scientist » függelékének kivonatai

A *GNU Free Documentation licence*-nek megfelelően (lásd 366. oldalon) a következő függelékeknek úgy, ahogy vannak kötelezően követniük kell az eredeti szöveg minden disztribúcióját, függetlenül attól, hogy azt módosították (lefordították például) vagy sem.

1.10.1 Contributor list

by Jeffrey Elkner

Perhaps the most exciting thing about a free content textbook is the possibility it creates for those using the book to collaborate in its development. I have been delighted by the many responses, suggestions, corrections, and words of encouragement I have received from people who have found this book to be useful, and who have taken the time to let me know about it.

Unfortunately, as a busy high school teacher who is working on this project in my spare time (what little there is of it ;-), I have been neglectful in giving credit to those who have helped with the book. I always planned to add an "Acknowlegdements" sections upon completion of the first stable version of the book, but as time went on it became increasingly difficult to even track those who had contributed.

Upon seeing the most recent version of Tony Kuphaldt's wonderful free text, "Lessons in Electric Circuits", I got the idea from him to create an ongoing "Contributor List" page which could be easily modified to include contributors as they come in.

My only regret is that many earlier contributors might be left out. I will begin as soon as possible to go back through old emails to search out the many wonderful folks who have helped me in this endeavour. In the mean time, if you find yourself missing from this list, please accept my humble apologies and drop me an email at jeff@elkner.net to let me know about my oversight.

And so, without further delay, here is a listing of the contributors:

Lloyd Hugh Allen

Lloyd sent in a correction to section 8.4. He can be reached at: lha2@columbia.edu

Yvon Boulianne

Yvon sent in a correction of a logical error in Chapter 5. She can be reached at: mystic@monuniverse.net

Fred Bremmer

Fred submitted a correction in section 2.1. He can be reached at: Fred.Bremmer@ubc.cu

Jonah Cohen

Jonah wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML. His Web page is jonah.ticalc.org and his email is JonahCohen@aol.com

Michael Conlon

Michael sent in a grammer correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters. Michael can be reached at: michael.conlon@sru.edu

Courtney Gleason

Courtney and Katherine Smith created the first version of horsebet.py, which is used as the case study for the last chapters of the book. Courtney can be reached at: orion1558@aol.com

Lee Harr

Lee submitted corrections for sections 10.1 and 11.5. He can be reached at: missive@linuxfreemail.com

James Kaylin

James is a student using the text. He has submitted numerous corrections. James can be reached by email at: Jamarf@aol.com

David Kershaw

David fixed the broken catTwice function in section 3.10. He can be reached at:

david_kershaw@merck.com

Eddie Lam

Eddie has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme. Eddie can be reached at: nautilus@yoyo.cc.monash.edu.au

Man-Yong Lee

Man-Yong sent in a correction to the example code in section 2.4. He can be reaced at: yong@linuxkorea.co.kr

David Mayo

While he didn't mean to hit us over the head with it, David Mayo pointed out that the word "unconsciously" in chapter 1 needed to be changed to "subconsciously". David can be reached at: bdbear44@netscape.net

Chris McAloon

Chris sent in several corrections to sections 3.9 and 3.10. He can be reached at: cmcaloon@ou.edu

Matthew J. Moelter

Matthew has been a long-time contributor who sent in numerous corrections and suggestions to the book. He can be reached at: mmoelter@calpoly.edu

Simon Dicon Montford

Simon reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13. He can be reached at: dicon@bigfoot.com

John Ouzts

John sent in a correction to the "return value" definition in Chapter 3. He can be reached at: jouzts@bigfoot.com

Kevin Parks

Kevin sent in valuable comments and suggestions as to how to improve the distribution of the book. He can be reached at: cpsoct@lycos.com

David Pool

David sent in a typo in the glossary of chapter 1, as well as kind words of encouragement. He can be reached at: pooldavid@hotmail.com

Michael Schmitt

Michael sent in a correction to the chapter on files and exceptions. He can be reached at: ipv6_128@yahoo.com

Paul Sleigh

Paul found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX. He can be reached at: bat@atdot.dotat.org

Christopher Smith

Chris is a computer science teacher at the Blake School in Minnesota who teaches Python to his beginning students. He can be reached at: csmith@blakeschool.org or smiles@saysomething.com

Katherine Smith

Katherine and Courtney Gleason created the first version of horsebet.py, which is used as the case study for the last chapters of the book. Katherine can be reached at: kss_0326@yahoo.com

Craig T. Snydal

Craig is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections, and can be reached at: csnydal@drew.edu

Ian Thomas

Ian and his students are using the text in a programming course. They are the first ones to test the

chapters in the latter half of the book, and they have make numerous corrections and suggestions. Ian can be reached at: ithomas@sd70.bc.ca

Keith Verheyden

Keith made correction in Section 3.11 and can be reached at: kverheyd@glam.ac.uk

Chris Wrobel

Chris made corrections to the code in the chapter on file I/O and exceptions. He can be reached at: ferz980@yahoo.com

Moshe Zadka

Moshe has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book. He can be reached at: moshez@math.huji.ac.il

1.10.2 Preface

by J. Elkner

This book owes its existance to the collaboration made possible by the internet and the free software movement. Its three authors, a college professor, a high school teacher, and a professional programmer, have yet to meet face to face, but we have been able to work closely together, and have been aided by many wonderful folks who have donated their time and energy to helping make it better.

What excites me most about it is that it is a testament to both the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

a) How and why I came to use Python

In 1999, the College Board's Advanced Placement Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum where I teach at Yorktown High School, in Arlington, Virginia. Up to this point, Pascal was the language of instruction in both our first year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first year course for the 1997-98 school year, so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first year class, I went looking for an alternative to C++.

A discussion on the High School Linux Users' Group mailing list provided a solution. A thread emerged during the latter part of January, 1999 concerning the best programming language for use with first time high school computer science students. In a posting on January 30th, Brendon Ranking wrote:

I believe that Python is the best choice for any entry-level programming class. It teaches proper programming principles while being incredibly easy to learn. It is also designed to be object oriented from its inception so it doesn't have the add-on pain that both Perl and C++ suffer from..... It is also *very* widely supported and very much web-centric, as well.

I had first heard of Python a few years earlier at a Linux Install Fest, when an enthusiastic Michael McLay told me about Python's many merits. He and Brendon had now convinced me that I needed to look into Python.

Matt Ahrens, one of Yorktown's gifted students, jumped at the chance to try out Python, and in the final two months of the 1998-99 school year he not only learned the language but wrote an application called

pyTicket which enabled our staff to report technology problems via the web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment combined with Matt's positive assessment of Python suggested Python was the solution I was looking for.

b) Finding a text book

Having decided to use Python in both my introductory computer science classes the following year, the most pressing problem was the lack of an available text book.

Free content came to the rescue. Earlier in the year Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational content. Allen had already written a first year computer science text book titled, **How to think like a computer scientist.** When I read this book I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming, rather than the features of a particular language. Reading it immediately made me a better teacher.

Not only was How to think like a computer scientist an excellent book, but it was also released under a GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version into the new language. While I would not have been able to write a text book on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational content.

Working on this book for the last two years has been rewarding for both me and my students, and the students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for errors in the book by giving them a bonus point every time they found or suggested something that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully, and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book actually sat in an unfinished state for the better part of a year until two things happened that led to its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our web site at http://www.ibiblio.org/obp called Python for Fun and was working with some of my most advanced students as a master teacher, guiding them beyond the places I could take them.

c) Introducing programming with Python

The process of translating and using How to think like a computer scientist for the past two years has confirmed Python's suitability to teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text dramatically illustrates this point. It is the traditional "hello, world" program, which in the C++ version of the book looks like this:

```
#include <iostream.h>
void main()
{
     cout << "Hello, world." << endl;
}
```

in the Python version it becomes:

```
print "Hello, World!"
```

Even though this is a trivial example, the advantages to Python stand out. There are no prerequisites to Yorktown's Computer Science I course, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The C++ version has always forced me to choose between two unsatisfying options: either to explain the #include, void main(), {, and } statements, and risk confusing or intimidating some of the students right at the start, or to tell them "just don't worry about all of that stuff now, we will talk about it later" and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to make their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing Section 1.5 of each version of the book, where this first program is located, further illustrates what this means to the beginning student. There are thirteen paragraphs of explanation of "Hello, world" in the C++ version, in the Python version there are only two. More importantly, the missing eleven paragraphs do not deal with the "big ideas" in computer programming, but with the minutia of C++ syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high level language like Python allows a teacher to postpone talking about low level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put "first things first" pedagogically.

One of the best examples of this is the way in which Python handles variables. In C++ a variable is a name for a place which holds a thing. Variables have to be declared with types at least in part because the size of the place to which they refer needs to be predetermined. Thus the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both Computer Science and Algebra). Bytes and addresses do not help the matter. In Python a variable is a name which refers to a thing. This is a far more intuitive concept for beginning students, and one which is much closer to the meaning of variable that they learned in their math class. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aides in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the key word def, so I simply tell my students, "when you define a function, begin with def, followed by the name of the function that you are defining, when you call a function, simply call (type) out its name." Parameters go with definitions, arguments go with calls. There are no return types or parameter types or reference and value parameters to get in the way, so I am now able to teach functions in less then half the time that it previously took me, with better comprehension.

Using Python has improved the effectiveness of our computer science program for all students. I see a higher general level of success and a lower level of frustration than I experienced during the two years using C++. I move faster with better results. More students leave the course with the ability to create meaningful programs, and with the positive attitude toward the experience of programming that this engenders.

d) Building a community

I have received email every continent on the globe and from as far away as Korea from people using this book to learn or to teach programming. A user community has begun to emerge and increasing numbers of people have been contributing to the project by sending in materials for the companion web site at http://www.ibiblio.org/obp.

With the publication of the book in print form, I expect the growth in the user community to continue and accelerate. It is the emergence of this user community and the possibility it suggests for similar collaboration among educators that has been the most exciting thing for me about working on the project. By working together we can both increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you.

Jeffrey Elkner
Yorktown High School
Arlington, Virginia

1.10.3 GNU Free Documentation License

Version 1.1, March 2000

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1.10.3.a.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document," below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical, or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, \LaTeX~input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary

formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

1.10.3.a.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1.10.3.a.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

1.10.3.a.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the

Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History," and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications," preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements," provided it contains nothing but endorsements of your Modified Version by various parties---for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1.10.3.a.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections

may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements," and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

1.10.3.a.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1.10.3.a.7 Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate," and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of Section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

1.10.3.a.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

1.10.3.a.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

1.10.3.a.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ

in detail to address new problems or concerns. See

http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Tartalomjegyzék