Bevezetés a Pythonba példákkal

Raphaël Marvie@lifl.fr

2005. július 6.

1.0.1 Verzió

Bevezetés

Egy kis történelem

A Python egy interpretált objektum orientált programozási nyelv. Az *Amoeba* (1990) operációs rendszer scriptnyelvéből származik. Guido van Rossum fejlesztette a CWI-n, az Amszterdami Egyetemen és a *Monthy Python's Flying Circus* -ról nevezte el.

Azóta a Python egy általános programozási nyelvvé vált *(a comp.lang.python -t 1994-ben hozták létre).*Komplett fejlesztő környezetet, egy versenyképes fordítót és nagy számú modult biztosít. Figyelemre méltó erőssége, hogy az elterjedt operációs rendszereken rendelkezésre áll.

A Python *nyílt forráskódú* nyelv, amit egy nagy közösség támogat és fejleszt : 300.000 felhasználó és több mint 500.000 letöltés évente.

Mire jó a Python ?

A Python erősségei négy pontban összefoglalva.

Minőség. A Pythonnal egyszerűen hozható létre scalable és könnyen karbantartható kód, az objektum orientált programozás előnyeit nyújtja.

Produktivitás. A Python azzal, hogy számos részletet az interpreter szintjére száműz, érthető kód gyors előállítását teszi lehetővé.

Portabilitás. Az, hogy az interpreter számos platformon rendelkezésre áll, lehetővé teszi ugyanannak a kódnak a futtatását akár egy PDA (Personal Digital Assistant)-n, akár egy nagy rendszeren[1].

Integráció. A Python tökéletesen alkalmas más programozási nyelveken írt (C, C++, Java Jython) komponensek integrálására. Egy interpreternek egy alkalmazásba való betöltése lehetővé teszi a Python scriptek integrálását a programba.

Néhány érdekes jellemző

- nincs explicit fordítási fázis
- nincs típusdeklaráció (deklaráció értékadáskor)
- automatikus memóriakezelés (hivatkozás számlálás)
- objektum orientált programozás
- dinamikus és interaktív
- bytekód generálásának lehetősége (a teljesítményt javítja az állandó interpretáláshoz

1 Még akkor is, hogyha ez egy kicsit bizarnak tűnik.

viszonyítva)

- standard interakciók (rendszerhívások, protokolok, stb.)
- integrálás a C és C++ nyelvvel

Python vagy nem Python

A többi programozási nyelvhez hasonlóan a Python sem a végleges megoldás minden igényre. Minden esetre számos környezetben alkalmazzák [4], mint a Google, a NASA, a zseb PC-k Opie környezetében, stb. A Python tehát «nagy alkalmazások» készítéséhez van adaptálva.

Mikor alkalmazzuk a Pythont ?Nagyon alkalmas gyors [1] és iterratív fejlesztésre. Lehetővé teszi a reaktivitást és a nehézkes programozás/fordítás/tesztelés ciklus elkerülését. Ráadásul a nyelv lehetővé teszi a kidolgozandó elméletek gyors ellenőrzését. Ha a megoldás már megszületett Pythonban, akkor mindig lehetőség van az optimalizálására egyes komponenseinek például C++ -ban történő újraírásával.
Lehetővé teszi, hogy nemcsak a fejlesztők, hanem a felhasználók által is egyszerűen bővíthető alkalmazásokat lehessen készíteni.
Végül egy kiváló jelölt a rendszer tasc-ok automatizálására. A Linox Gentoo disztribúciójának *portage* rendszer adminisztrációját Pythonban fejlesztették.

Mikor ne használjunk Pythont ? A dinamikus volt ritkán párosul nagy teljesítménnyel, azaz a Python biztosan nem a legmegfelelőbb nyelv az intenzív adatkezelésre.
Bizonyos esetekben a felhasználók a Python alkalmazását a komponensek integrációjára korlátozzák azért, hogy versenyképes rendszereket hozzanak létre. A pythonos változat képezi az elképzelés igazolását illetve a program egy működő példányát, mielőtt a program gyengébb részekit más nyelven újraírnák.
Végezetül egy dolog bizonyos: a Python lehetővé teszi a rossz elképzelések gyors azonosítását ...

1. Első lépések

1.1. A Python használata

A Python a scriptnyelvek többségéhez hasonlóan mind interaktív, mind script módban használható. Az első esetben a felhasználó és az interpreter között párbeszéd van: a felhasználó által beírt parancsok kiértékelése au fur et á mesure történik. Ez nagyon praktikus prototípus készítéskor, egész programok vagy programrészek tesztelésekor, vagy komplex adatstruktúrák manipulálása esetében. A következő lista bemutatja az interpreter elindítását (gépeljük a shell prompt[2] után: **python**) és az interpreterből való kilépést (gépeljünk **Ctrl-D** -t).

```
$ python
Python 2.3.5 ( #2, Feb 9 2005 , 11:52:33)
[GCC 3.3.5 ( Debian 1:3.3.5 -8)] on linux2
Type " help " , " copyright " , " credits " or " license " for more information
.
>>> print 'hello world !'
hello world !
>>> ^D
$
```

Az interpreter script módú használatakor a kiértékelendő utasításokat file-ba mentjük, mint bármelyik program esetében. Ez esetben a felhasználónak az általa használt szövegszerkesztővel be kell írni az összes utasítást, majd az nterpretertől kérnie kell a végrehajtásukat. A python-file-okat a .py kiterjesztés azonosítja. A következő listában az interpretert egy programnév paraméterrel hívjuk (a hagyományos 'hello world' -del).Unix rendszer esetén a `hello.py` file első sora megadja, hogy melyik interpretert kell a file kiértékelésére használni, ha a file végrehajtható. Ebben az esetben elegendő begépelni a file nevét a shell prompt után.

```
$ cat hello .py
#! / usr/bin/env python
print 'hello world !'
$ python hello .py
hello world !
```

2 Az egész jegyzetben a shell prompt-ot $ -lal jelöm.

1.2. Alapstruktúrák

1.2.1 Kommentek

A scriptnyelvek többségéhez hasonlóan a Python kommenteket a # karakter definiálja. A # karaktertől kezdődően az interpreter a sor végéig kommentként értelmezi a szöveget. Mint mindig, használjuk a kommenteket bőségesen. Ne habozzunk a kódunkat kommentezni ! A következő lista egy kommentsor. Az 1.3.5 fejezetben újra megvizsgáljuk a kommenteket az öndokumentálás vonatkozásában. A # -gal bevezetett kommenteket a kód kivitelezésére vonatkozóan kell fenntartani.

```
>>> # ez egy komment
>>> print 'bouh ' # ez is egy komment
bouh
```

1.2.2 Típusadás a Pythonbam

A Pythonban minden objektum. Bármilyen adatokat is manipulálunk, ezek az adatok objektumok, amiknek az osztályait a felhasználó, vagy alaptípusok esetében a Python-környezet definiál. Az egyik következmény az, hogy a felhasználó az adatokat referenciákkal manipulálja (amik a példányok funkcióihoz adnak hozzáférést). Ez a megközelítés a felhasználó által manipulált adatok homogén visszaadását teszi lehetővé (a SmallTalk -hoz hasonlóan). Ennek megfelelőn a következő adatok valamennyien objektumok: *1, [2, 3, 4], 5.6, 'toto', a Foo egy példánya.*

A Python egy dinamikus típusadású nyelv. Ez nem azt akarja jelenteni, hogy azoknak az adatoknak, amiket manipulálunk nincs típusuk, hanem azt, hogy a típusuk meghatározására a használatukkor kerül sor[3]. Ebben az összefüggésben a változók típusát a felhasználó nem definiálja explicit módon. Ezért ugyanaz a változó egy programon belül különböző típusú[4] objektumokra hivatkozhat.

```
>>> x = 1 # x reference un entier
>>> x = 'toto ' # x reference desormais une chaine
>>> x = Foo () # x reference desormais une instance de Foo
```

1.2.3 Aritmetika

A Python az aritmetikai műveletek rendkívül egyszerű kifejezését teszi lehetővé. Abben az esetben, amikor minden egész, akkor az eredmény is egész. Ha az operandusok közül legalább az egyik valós

3 A Pythonban , ce calcul se résume á la possibilité pour l'objet de recevoir un message particulier.
4 Ezt az adottságot csak a polimorfizmus örve alatt kellene használni, hogy a program olvashatósága ne romoljon.

típusú, akkor a Python az összes operandust valós típusúvá alakítja. A következő litában a baloldali oszlop egészeken, a jobboldali oszlop valós számokon végzett elemi műveleteket mutat be:

>>> x = 1 + 2
>>> y = 5 * 2
>>> y / x
3
	>>> y = 5.5 * 2
	>>> y
	11.0

>>> y % x
2
	>>> *x = 12.0 / 3*
	>>> *x*
	4.0

Az aritmetika vonatkozásában az értékadásnak két formája lehet. Ennek a két formának különböző az értelme. A következő lista két értékadást mutat, amiket azonos módon érthetnénk. Minden esetre az első forma (2. sor) következménye egy új, egész típusú változópéldány létrehozása, ami arra való, hogy az x változó 2 -vel növelt értékét tartalmazza. A második forma (3. sor) új változópéldány létrehozása nélkül ad x értékéhez 2-t.

```
>>> x = 4
>>> x = x + 2
>>> x += 2
```

1.2.4 Karakterláncok

A karakterláncokat különböző módokon definiálhatjuk a Pythonban. Az egyszeres és a kettős idézőjel használatában nem tesz különbséget. A választást gyakran a karakterlánc tartalma szabja meg : egyszeres idézőjelet tartalmazó stringet kettős idézőjellel deklarálunk és viszont. A többi esetben az idézőjel fajtája indifferens. A Pythonban minden objektum, így a karakterláncok is azok. A következő lista két karakterláncot definiál, amikre az ***x*** és ***y*** -nal hivatkozunk. A ***z*** -vel hivatkozott string többsoros (három egyszeres vagy kettős idézőjel használata).

```
>>> x = 'hello '
>>> y = " world !"
>>> z = '''hello
world '''
```

1.2.4.1 Konkatenáció

Karakterláncokat kétféle módon kapcsolhatunk össze. Mindkét esetben a + operátort alkalmazzuk a konkatenálásra. A jobboldali forma egy rövidebb írásmód.

```
>>> x = x + y
>>> x
'hello world !'
```

```
>>> x += y
>>> x
'hello world !'
```

1.2.4.2 Kiíratás

A karakterláncok kiíratása a print utasítással történik úgy, hogy a stringeket explicit módon összekapcsoljuk (vagy a konkatenáció operátorát, vagy vesszőket alkalmazunk) vagy egy formátumstringet alkalmazunk, mint a C nyelv **printf** függvénye. Ez utóbbi opció hatékonyabb, de nehezebb a használata. A következő lista a karakterláncok háromféle kiíratási módját mutatja be.

```
>>> print 'I say : ' + x
I say : hello world !
>>> print x , 2 , 'times '
hello world ! 2 times
>>> print "I say : % s %d time (s)" % (x , 2)
I say : hello world ! 2 time (s)
```

1.2.4.3 Manipulációk

A Pythonban nagyon egyszerűen férhetünk hozzá egy karakterlánc karaktereihez: a stringet egy indexelt karakter szekvenciaként manipuláljuk. Így mindegyik karakterhez közvetlenül hozzáférhetünk - zárójelek alkalmazásával - az indexe segítségével (az első karakter indexe 0). A karakterekhez történő egyedi hozzáférésen túl al-stringekhez is hozzáférhetünk a kívánt stringrészlet kezdő indexének (ami beleértendő) és végindexének (ami nem értendő bele az al-stringbe) megadásával, amiket **:** választ el egymástól. Az al-stringek esetében nem az eredeti string egy részletéhez férünk hozzá, hanem kapott érték annakegy másolata.

A következő lista néhány példát ad arra, hogyan férhetünk hozzá egy karakterhez (2. sor), illetve al-stringekhez (a többi sor). A baloldali oszlop azt mutatja be, amikor a string eleje felől férünk az al-stringhez (az indexek pozitívak). A 6. sor azt jelenti, hogy x -hez a string 4. karakterétől a string végéig terjedő al-stringet akarjuk hozzárendelni. A jobboldali oszlop olyan példákat mutat be,

melyekben egyes indexeket a string végéhez képest adunk meg (ezek negatívak). Végül az utolsó sor az x karakterlánc másolatát készíti el.

```
>>> x = 'hello world !'
>>> x[4]
'o'
>>> x [2:4]
'll '
>>> x [3:]
'lo world !'
>>> x[:]
'hello world !'
```

```
>>> x[ -3:]
'ld!'
>>> x[1: -1]
'ello world '
```

1.2.5. Listák

A Python listák elemek rendezett együttesei. Ezek az együttesek különböző típusú elemeket tartalmazhatnak, egyetlen közös tulajdonságuk, hogy objektumok. Mivel mindegyikük objektum, a listák maguk is objektumok (a ***list*** osztály objektumai). A következő példa egy True értéket tartalmazó változót hoz létre, utána egy **`foo`** nevű listát, ami a 'bar' stringet, az 12345 egészet és az x változó tartalmát tartalmazza.

```
>>> x = True
>>> foo = [ 'bar ' , 12345 , x]
```

A lista végéhez az **`append()`** metódussal, egy adott indexű helyéhez az **`insert()`** metódussal adhatunk elemeket. Az **`extend()`** metódus a paramétereként megadott lista tartalmát hozzáadja a listához.

```
>>> foo . append ( 'new ')
>>> foo
['bar ' , 12345 , 1 , 'new ']
>>> foo . insert (2 , 'new ')
>>> foo
['bar ' , 12345 , 'new ' , 1, 'new ']
>>> foo . extend ([67 , 89])
>>> foo
['bar ' , 12345 , 'new ' , 1, 'new ' , 67 , 89]
```

Az **`index()`** metódus megadja egy elem listabeli első előfordulásának indexét. Abban az esetben, ha a paraméterként megadott elem nincs a listában, egy **`ValueError`** generálódik. Az **`in`** utasítás **`True`** értéket ad vissza, ha az elem a listában van, egyébként **`False`** -ot.

```
>>> foo . index ( 'new ')
2
>>> foo . index (34)
Traceback ( most recent call last ):
File "<stdin >" , line 1 , in ?
ValueError : list . index (x): x not in list
> > > 34 in foo
False
```

A listákban és a karakterláncokban az a közös, hogy rendezett együttesek. A listák elemeihez is

indexeléssel férhetünk hozzá (1. sor). Lehetőség van a lista egy részének kinyerésére, illetve a lista másolatának előállítására (5. sor). Ez utóbbi eljárás azért nagyon fontos a listák bejárása során (lásd 1.4.2 fejezetet), hogy ne módosítsuk a bejárt listát.

```
>>> foo [2]
'new '
>>> foo [1: -3]
[12345 , 'new ' , 1]
>>> bar = foo [:]
>>> bar . append (3)
>>> foo [ -1:]
[89]
>>> bar [ -1]
3
```

A listák konkatenációval egyesíthetők. Ezeket a konkatenációkat másolással és hozzáadással is megvalósíthatjuk. Az 1. sor két létező lista másolásával megvalósított konkatenációt mutat be. A 4. sor egy másolat, amit elemeknek egy létező listához történő konkatenálásával valósítunk meg. Végül a 7. sor egy listának egy minta (ami egy allista) ismétlésével való létrehozását mutatja be.

```
>>> bar = [0 , 1] + [1 , 0]
>>> bar
[0 , 1 , 1 , 0]
>>> bar += [2 , 3]
[0 , 1 , 1 , 0 , 2 , 3]
>>> bar
> > > [0 , 1] * 3
[0 , 1 , 0 , 1 , 0 , 1]
```

Ahhoz hasonlóan, ahogyan elemeket adunk egy listához, törölhetünk is elemeket egy listából. A **`remove`** metódus segítségével törölhetjük a megadott elem első elpfordulását egy listából (1. sor). Ha a paraméterként megadott elem nem fordul elő a listában, akkor egy **ValueError** kivétel (exeption) generálódik (4. sor). A Pythonban minden objektum, így egy részlista is, a **del** (delete) operátorral rombolható le (8. sor).

```
>>> foo . remove ( 'new ')
>>> foo
['bar ' , 12345 , 1 , 'new ' , 67 , 89]
>>> foo . remove (34)
```

```
Traceback ( most recent call last ):
File "<stdin >" , line 1 , in ?
ValueError : list . remove (x): x not in list
>>> del foo [1:3]
>>> foo
['bar ' , 'new ' , 67 , 89]
```

A ***lista mapping*** lehetővé teszi, hogy egy lista minden egyes elemére alkalmazzunk egy operációt. A 2. sor azt jelenti, hogy a lista minden elemét megszorozzuk 2-vel[5].

1.2.1. Listák és karakterláncok

A listák és a karakterláncok hasonlóak a struktúrájuk és a rajtuk végezhető műveletek tekintetében. Bizonyos – a karakterláncokra rendelkezésre álló – metódusok a két adatstruktúrát manipulálják.
A karakterláncoknál rendelkezésünkre álló **`join`** metódus lehetővé teszi egy karakterlánc stringlistákból történő létrehozását. Azt a stringet, amire a **`join`** metódust hívjuk, a lista különböző elemei közötti szeparátorként használjuk.

```
>>> ' ; ' . join ([ 'a' , 'b' , 'c'])
'a ; b ; c'
```

Szimmetrikusan : a stringeknél rendelkezésünkre álló **`split`** metódus lehetővé teszi egy karakterlánc rész-stringekre történő darabolását. Ez a feldarabolás egy, vagy több karakter szerint történhet (1. sor). Abban az esetben, amikor második argumentumként egy egészet adunk meg, ez az egész a rész-stringek maximális számát adja meg.

```
>>> 'hello crazy world !' . split (" ")
['hello ' , 'crazy , 'world ']
>>> 'hello crazy world !' . split (" " , 1)
['hello ' , 'crazy world !']
```

1.2.2. Tuplek

A **`tuplek`** megváltoztathatatlan (immutable) elemek rendezett együttesei. A listákhoz hasonlóan a tuplek is különböző típusú elemeket tartalmazhatnak. Az 1. sor egy klasszikus – zárójelekkel való – deklarációt mutat be, míg a 2. sor a rövidített jelölést mutatja be. A vessző azért lényeges, mert az

5 Ez a konstrukció nagyon hasonlít a Lisp által biztosított funkcionális programozáshoz.

határozza meg, hogy egy egyelemű listáról, nem pedig a 12 értékről beszélünk. Ez a jelölésmód egy egy elemű tuple zárójel nélküli deklarációjánál lesz lényeges.

```
>>> foo = ( 'bar ' , 12345 , x)
>>> bar = 12 ,
```

A listákhoz hasonlóan a tuplek elemeihez is indexeléssel férhetünk hozzá (1. és 3. sor) és az **`in`** konstrukcióval tesztelhetjük egy elem egzisztenciáját a listában. Ha már létrehoztunk egy listát, akkor annak a tartalma többé nem változtatható meg.

```
>>> foo [1]
12345
>>> foo [: -1]
('bar ' , 12345)
>>> 'bar ' in foo
True
```

Megjegyzés. Egy lista illetve egy tuple közötti választás kritériumai :

- a tuplek bejárása gyorsabb, mint a listáké
- konstansok definiálására használjunk tupleket

A tuplek átalakíthatók listákká (1. sor) és viszont (3. sor).

```
>>> list ((1 , 2 , 3))
[1 , 2 , 3]
>>> foo = tuple ([1 , 2 , 3])
>>> foo
(1 , 2 , 3)
```

Végül, a Python lehetőséget ad arra, hogy egy tuple-ból szimultán különböző értékeket rendeljünk változókhoz (1. sor). Ez esetben megint elhagyhatók a zárójelek (4. sor.)

```
>>> (x , y , z ) = foo
>>> x
1
>>> a , b , c = foo
>>> b
2
```

1.2.3. Szótárak

A szótárak – néha asszociatív tömböknek is nevezik őket – kulcsokkal indexelt, nem rendezett adategyüttesek. Egy kulcsnak kötelezően megváltoztathatatlannak (immutábilisnak) (string, egész vagy tuple) kell lenni. Másrészt egy kulcs mindig egyedi. Egy üres szótárat két kapcsos zárójellel definiálunk (1. sor). Elemadás egy szótárhoz (2. és 3. sor), illetve hozzáférés egy szótár elemeihez (6. sor) az elemhez rendelt kulccsal való indexeléssel történik. Egy szótár kiíratásakor (5. sor) egy « kulcs : érték » párokból álló lista jelenik meg.

```
>>> mydict = {}
>>> mydict [ 'foo '] = 456
>>> mydict [123] = 'bar '
>>> mydict
{123: 'bar ' , 'foo ': 456}
>>> mydict [123]
'bar '
```

A szótáraknak vannak a kulcsok manipulálására szolgáló metódusai. A **`keys`** metódus (1. sor) a szótár kulcsainak listáját adja visszatérési értékként, a **`has_key`** metódus **`True`** értéket ad vissza, ha a paraméterként megadott kulcs szerepel a szótárban, ellenkező esetben **`False`** -ot.

```
>>> mydict . keys ()
[123 , 'foo ']
>>> mydict . has_key ( 'bar ')
False
```

A **`values()`** metódussal egy szótárban tárolt értékeket tartalmazó listához férünk hozzá (1. sor). Az items() metódus egy tuplekből álló listát ad, melyben midegyik tuple egy kulcs-érték párt tartalmaz (3. sor).

```
>>> mydict . values ()
['bar ' , 456]
>>> mydict . items ()
[(123 , 'bar ') , ( 'foo ' , 456)]
```

Egy kulcshoz asszociált értéket úgy módosítunk, hogy újra értéket rendelünk a szótárban a kérdéses indexhez (1. sor). Végul a **`del`** operátorral törölhetünk egy kulcs-érték párt a szótárból (4. sor).

```
>>> mydict [123] = 789
```

```
>>> mydict
{123: 789 , 'foo ': 456}
>>> del mydict [ 'foo ']
>>> mydict
{123: 789}
```

A kulcs- és értékmanipuláló metódusok segítségével többféle lehetőségünk van a szótárok bejárására. A következő néhány sor két példát ad a bejárásra.

```
>>> flames = { 'windows ': 'bof ' , 'unix ': 'cool '}
>>> for key in flames . keys ():
... print key , 'is ' , flames [ key]
...
windows is bof
unix is cool
>>> for key , value in flames . items ():
... print key , 'is ' , value
...
windows is bof
unix is cool
```

Az utolsó példa a lista mapping és egy formátumstring (aminek az alkalmazása nem korlátozódik a kiíratásra) segítségével történő bejárást mutat be.

```
>>> bar = { 'a': 'aa ' , 'b': 'bb ' , 'c': 'cc '}
>>> ["%s=%s" % (x , y) for x , y in bar . items ()]
['a=aa ' , 'c=cc ' , 'b=bb ']
```

1.3. Szerkezetek

1.3.1. Struktúrálás és behúzás

Egy Python program struktúráját a behúzások definiálják. Egy kódblokk elejét egy „**:**” definiál, az első sort fejlécnek tekinthetjük (teszt, ciklus, definíció, stb.). A fejléchez képpest a kódblokk teste beljebb van igazítva (de azonos mértékben). A kódblokk ott ér véget, ahol az utasítássor behúzása a fejléc behúzásával megegyezik. (Ez utóbbi utasítássor már nem tartozik az előző kódblokkhoz.) A kódblokkok egymásba ágyazhatók.

```
<fejléc >:
      <utasítások >
```

Kisméretű, pl. egy utasításból álló blokk esetében a blokkot egyetlen sorban definiálhatjuk. A '**:**' karakter mindig a blokk fejlécének a blokk testétől való elhatárolására szolgál. A blokk testében lévő utasításokat '**;**' választja el egymástól[6].Azonban ez a használatmód nem mindig előnyös amikor a kód olvashatóságáról van szó, ezért kerülendő[7].

```
>>> < fejléc >: < utasítás > ; < utasítás >
```

Ez a struktúrálás ugyanilyen jól használható ciklusok, tesztek, függvények, osztályol vagy metódusok definiálására.

1.3.2. Tesztek

Logikai feltételek A Pythonban minden ami nem hamis: igaz. Úgy foglalható össze, hogy az „üres” adatok : **`False, 0, "", [], {}, (), None`** fals-ok.

A Python egyetlen konstrukciót kínál tesztek készítéséhez: az **`if then else`** konstrukciót. Ennek egy sajátossága a tesztek láncolásának lehetősége az **`elif`** konstrukcióval.

```
if x == 'hello ':
print 'hello too!'
elif x == 'bonjour ':
print 'bonjour aussi !'
else :
print 'moi pas comprendre '
```

Szekvenciák (stringek, listák és tuplek) összehasonlítása a lexikográfiai sorrendjük alapján történik. A következő két teszt igaz logikai értékű.

```
(1 , 2 , 3) < (1 , 2 , 4)
(1 , 2 , 3) < (1 , 2 , 3 , 4)
```

1.3.3. Ciklusok

A Pythonban két fajta ciklus van : a számlált ciklusok (**`for`**) és a hátul tesztelésen alapuló (**`while`**) ciklusok. Ez a két konstrukció azonos sémát követ: egy fejlécből áll, ami leírja a ciklus menetét; egy utasítás együttesből, aminek a végrehajtására ciklus minden egyes fordulójában sor kerül és egy

6 A ';' -k használata a sorok végén opcionális a Pythonban. A ';' -t mindig felhasználhatjuk az egyazon sorban lévő utasítások elválasztására.

7 Az az idő már elmúlt, amikor számított a karakterek file-beli száma.

opcionális (ezt az **else** kulcsszó vezeti be) részből, aminek kiértékelésére a ciklusból történő kilépéskor kerül sor. Végül, mint a C-ben a ciklusok tartalmazhatnak **continue** elágazásokat, amik a következő iterrációs ciklusra való áttérésre szolgálnak és **break** elágazásokat, amik a ciklusból való kilépésre szolgálnak (ebben az esetben nem kerül sor az **else** klauzula kiértékelésére).

1.3.3.1 for ciklusok

Egy **for** ciklus definiál egy változót, ami egymás után fölveszi a bejárt szekvencia (lista vagy tuple) valamennyi értékét (1. sor).Az **else** klauzula kiértékelésére akkor kerül sor, ha a szekvencia elfogyott és nem volt break (3. sor).

```
for <var > in < szekvencia >:
      < utasítások >
else :
      <utasítások , a szekvencia elfogyott és nem volt break >
```

A **range()** függvény egy alsó és felső határok közé eső, egész számokból álló listát állít elő. Ez a konstrukció akkor hasznos, amikor egy **for** ciklus egy egész számokból álló szekvenciát jár be. A **range()** függvény háromféle módon használható :

- egy paraméterrel, ami megadja az elemek számát (1. sor)
- két paraméterrel, ami megadja az alsó és a felső határt (az alsó határ benne van a listában, a felső határ nincs benne) (3. sor)
- három paraméterrel, amik megadják a határokat és a lépésközt (a szekvencia két szonszédos eleme közötti növekményt) (5. sor)

```
>>> range (6)
[0 , 1 , 2 , 3 , 4 , 5]
>>> range (3 , 7)
[3 , 4 , 5 , 6]
>>> range (0 , 10 , 3)
[0 , 3 , 6 , 9]
```

Egy karakterláncot bejáró (2-3. sor) és egy listát bejáró (7-8. sor) **for** ciklus.

```
>>> a = [ 'hello ' , 'world ']
>>> for elt in a:
```

```
        print elt
...
hello
world
>>> for idx in range ( len (a)):
        print idx , a[idx ]
...
0 hello
1 world
```

1.3.3.2 `while` ciklusok

Egy while ciklust egy logikai feltétel definiál ugyanolyan szabályokkal, mint amik a tesztekre érvényesek (1. sor). Amíg ez a feltétel teljesül, addig a **`while`** -hoz kapcsolt blokk utasításai értékelődnek ki. Az **`else`** klauzula kiértékelésére akkor kerül sor, ha a feltétel hamis és nem volt break (3. sor).

```
>>> while 1:
...        pass
...
Traceback ( most recent call last ):
     File "<stdin >" , line 1 , in ?
KeyboardInterrupt
```

1.3.4. Függvények

A Python-ban csak függvények vannak. Egy függvényt a **`def`** kulcsszó definiál. Egy függvénynek mindig van egy visszatérési értéke. Ha egy függvény nem tartalmazza a **`return`** klauzulát, akkor a **`None`** a visszatérési értéke[8].

```
>>> def fib (n): # suite de fibonacci jusque n
…        a , b = 0 , 1
...      while b < n:
...          print b,
...          a , b = b , a + b
...
>>> fib (100)
```

8 Ebben az esetben esetleg beszélhetünk eljárásról.

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

Egy függvény paraméterei definiálhatók alapértelmezett értékekkel. A függvény használatakor ezek az értékek tehát opcionálisak. Egy opcionális paraméterekkel rendelkező függvény esetében a paraméterek alkalmazásának rendezettnek kell lenni, a paraméterek értékadása a definíciójuk sorrendjében (6.sor) vagy a megnevezésük segítségével történik (8.sor).

```
>>> def welcome ( name , greeting ='Hello ' , mark ='!'):
...      print greeting , name , mark
...
>>> welcome ( 'world ')
Hello world !
>>> welcome ( 'monde ' , 'Bonjour ')
Bonjour monde !
>>> welcome ( 'world ' , mark ='... ')
Hello world ...
```

A Python a *lambda*[9] formula alkalmazásával lehetővé teszi anonym függvények definiálását. Egy ilyen függvény egyetlen kifejezésre korlátozódik. Ez a konstrukció arra alkalmas, hogy függvényeket más függvények konfigurálásához paraméterekként adjunk meg vagy a list mapping esetében alkalmazható (lásd az 1.2.5 fejezetet).

```
>>> def compare (a , b , func =( lambda x,y: x < y)):
...      return func (a,b)
...
>>> compare (1 , 2)
1
>>> compare (2 , 1 , func =( lambda x,y: x > y))
1
```

9 Ici encore, cela sent la programmation fonctionnelle type Lisp `a plein nez. . .

1.3.5. Dokumentálás

A dokumentálás a kód integráns részét képezi. A **#** segítségével történő dokumentálásból adódóan a Python öndokumentáló kódok írásának lehetőségét kínálja. Ez egy értékes lehetőség a függvények, osztályok és modulok esetében. Ezeknek a különféle elemeknek az ilyen fajta dokumentációihoz az illető elemek **__doc__** attribútuma segítségével férhetünk hozzá. Ez a technika automatikusan alkalmazható az integrált fejlesztő környezetek használatával vagy interaktív módban a **help** funkció használatával.

A **#** -gal definiált kommenteket a kódra vonatkozó technikai megjegyzések és a kérdéses elem fejlesztői számára kell fenntartani. A dokumentáció a fejlesztőknek, de még inkább a dokumentált elem felhasználójának van szánva. Tehát tükröznie kell az elem viselkedését. Ez a második forma párhuzamba van állítva a *javadoc* -kal.

A dokumentáció definiálása a deklaráció utáni első sorban egy többsoros karakterlánc definíció segítségével történik[10] (2-5. sor). A dokumentáció definiálásának egy jó módszere a következő : adunk egy egysoros rövid leírást, kihagyunk egy sort és leírjuk a részleteket.

```
>>> def dummy ():
...      '''Ez a függvény nem egy nagy durranás ...
...
...      Ez a függvény valóban nem egy nagy durranás ,
...      semmit se csinál .'''
...      pass
...
>>> help ( dummy )
Help on function dummy in module __main__ :
dummy ()
          Ez a függvény nem egy nagy durranás ...
          Ez a függvény valóban nem egy nagy durranás ,
          semmit se csinál .
```

1.1. A szekvenciákra vonatkozó egyéb elemek

1.1.1. Szekvenciák manipulálása

A Pythonnak számos olyan függvénye van, amit a szekvenciák minden elemére alkalmazhatunk

10 Még akkor is, ha a karakterlánc csak egy sort foglal el.

annak érdekében, hogy szűrőket definiáljunk, számításokat végezzünk mindegyik elemen vagy hogy hashkódokat számoljunk.

1.1.1.1 filter

A filter az első argumentumként megadott függvényt alkalmazza a második argumentumként megadott szekvencia minden egyes elemére és visszatérési értékként egy olyan új listát ad, ami a szekvencia összes elemét tartalmazza, melyekre a függvény visszatérési értéke true.

```
>>> def funct1 ( val ):
...       return val > 0
...
>>> filter ( funct1 , [1 , -2 , 3 , -4 , 5])
[1 , 3 , 5]
>>> def iseven (x):
...       return x % 2
...
>>> filter ( iseven , [1 , 2 , 3 , 4 , 5 , 6])
[1 , 3 , 5]
```

1.1.1.2 map

A **`map`** függvény az első argumentumként megadott függvényt hívja az argumentumként megadott szekvencia vagy szekvenciák mindegyik elemére. Ha több szekvenciát adunk meg argumentumként, akkor a függvénynek annyi paraméterrel kell rendelkezni, ahány szekvenciát argumentumként megadtunk. A map egy listát ad visszatérési értékként, ami az egyes számítások eredményeit tartalmazza.

```
>>> def sum (x , y):
... return x + y
...
>>> map ( sum , [1 , 2 , 3] , [4 , 5 , 6])
[5 , 7 , 9]
>>> map ( iseven , [1 , 2 , 3 , 4 , 5 , 6])
[1 , 0 , 1 , 0 , 1 , 0]
```

1.1.1.3 reduce

A **`reduce`** függvény egy szekvenciát redukál úgy, hogy annak mindegyik elemére rekurzívan alkalmaz egy függvényt. Az első paraméterként megadott függvénynek két argumentuma kell, hogy legyen. A **`reduce`** függvénynek lehet egy opcionális harmadik paramétere is, ami a rekurzív számolás kezdőértéke.

```
>>> reduce ( sum , [1 , 2 , 3 , 4 , 5])
15
>>> reduce ( sum , [1 , 2 , 3 , 4 , 5] , -5)
10
```

1.1.2. Problémás `for` ciklusok

Egyes listamanipuláló ciklusok problémásak lehetnek, például ha a ciklus módosítja azt a listát, amit éppen bejár. Ahhoz, hogy az ilyen fajta ciklus rendben lefusson és ne legyen végtelen ciklus, bizonyos esetekben lényeges, hogy a lista bejárásához készítsünk a listáról egy másolatot és az eredeti listát módosítsuk. Alapértelmezetten minden argumentumátadás hivatkozással és így másolatkészítés nélkül történik (valójában a hivatkozások listáját, nem pedig objektumokat adjuk át). A következő ciklus végtelen ciklus lenne.[11]

```
>>> a = [1 , -2 , 3 , -4 , 5 , -6]
>>> for elt in a [:]:
... if elt > 0: a. insert (0 , elt)
...
>>> a
[5 , 3 , 1 , 1 , -2 , 3 , -4 , 5 , -6]
```

11 Ez a ciklus pozitív elemeket ad a lista elejéhez. Mivel a lista első eleme pozitív, ezért a ciklus mindig az 1 értékű elemnél fog tartani, a lista pedig nő.

2. Néhány modul és *built-in*

2.1. Modulok definíciója és használata

2.1.1. Definíció

Az előző fejezetben a kódrészleteket interaktívan írtuk be. Ez nem életképes módszer, ha a kódot többször, vagy több gépen akarjuk végrehajtatni. Ahhoz, hogy a kód ne vesszen el az első megoldás az, hogy írunk egy « programot », ami azt jelenti, hogy a kódot egy .py kiterjesztésű szövegfileba írjuk. Egy program többször végrehajtható. Azonban még ha ez a program függvényekkel megfelelően is van struktúrálva, a függvények újra felhasználása nem olyan egyszerű (hacsak nem copy-paste -tel, ami egy borzalom).
A takarékosabb fejlesztés érdekében a Python a modul fogalmát kínálja. Egy modul lehetővé teszi programokba integrálható függvénykönyvtárak, adatstruktúrák, osztályok szolgáltatását. A Python esetében egy modul létrehozása egy program írását jelenti : létrehozunk egy file-t. Egy file-ban található definíciók globálisan és egyedileg is felhasználhatók. Így az **`examples.py`** felhasználható egy modulként (amit ugyanúgy nevezünk, mint a file-t) és így a két függvényt hozzáférhetővé tesszük a modult használó valamennyi program számára[12].

A dokumentációs stringnek meg kell előzni az összes deklarációt (tehát az **`import`** klauzulákat is) ahhoz, hogy a modul dokumentációjának tekintsük. E file első két (opcionális) sorának a következő a jelentése :

- a file első sora jelzi, hogy egy Unix környezet ezt a file-t Python programként ismeri fel (ha mint végrehajtható file-t definiáljuk)[13]
- a második sor a file kódolását definiálja, esetünkben a standard nyugat-európai kódolást, ami támogatja az ékezetes karakterek használatát. Ha egy Python-file ékezetes karaktert tartalmaz, akkor ennek a sornak kötelezően szerepelni kell a scriptben (ellenkező esetben kihagyható).

12
13 Unix alatt a #! konstrukció annak megadására szolgál, hogy egy végrehajtandó scriptről van szó

```
#! /usr/bin/env python
# -*- coding: iso-8859-1 -*-
#
# examples.py
#
"""
Regroupe les définitions des fonctions relatives au chapitre 1 de
‘Initiation a python par l’exemple’.
"""
def fib (n):
     ’’’Calcule la suite de Fibonacci jusque n’’’
       a, b = 0, 1
       while b < n:
            print b,
            a, b = b, a + b
def welcome (name, greeting=’Hello’, mark=’!’):
     ’’’Un hello world configurable’’’
     print greeting, name, mark
```

Modul használata Az **`import`** konstrukció teszi lehetővé egy modul importálását és a tartalmához való hozzáférést[14]. Egy modult kétféle módon importálhatunk. Az első megoldás az, hogy megjelöljük a használni kívánt modult, tehát a tartalma « minősített névmegadással » férhető hozzá, azaz a függvény neve elé a modul nevét prefixumként oda kell írni (2. sor). A második megoldás a **`from import`** konstrukciót alkalmazza. Ebben az esetben megadjuk azokat a függvényeket, amiket importálni akarunk egy modulból (4. sor[15]). Tehát a modulnevet nem használjuk prefixumként (nem alkalmazunk minősített névmegadást).Végül ennek a második módszernek az esetében a '*' jelölés alkalmazásával lehetőségünk van egy modul valamennyi elemének az importálására (7. sor).

```
>>> import examples
>>> examples . welcome ( rafi )
Hello rafi !
>>> from examples import welcome
>>> welcome ( rafi )
Hello rafi !
>>> from examples import *
>>> fib (100)
1 1 2 3 5 8 13 21 34 55 89
```

14 Itt feltételezem, hogy az interpretert az examples.py file-t tartalmazó könyvtárban indítottuk el.
15 Több függvény importálása esetén a függvényneveket vesszővel választjuk el egymástól.

Kevert alkalmazás Egy file-t egyidejűleg difiniálhatunk modulként és programként. Ez a kevert definíció például azért érdekes, mert egy program funkcióit a programtól függetlenül használhatjuk, vagy tesztkódot asszociálhatunk egy modulhoz. Ebben az esetben a fájl vége a kódinicializáló definíciót mint program tartalmazza. Egy tesztet hajtunk végre annak érdekében, hogy megtudjuk: a kód importálva van-e, vagy végrehajtandó. Ha a kód importálva van, akkor a **__name__** változó a modul nevét tartalmazza, ha végrehajtandó akkor a **__main__** stringet (1. sor). A végrehajtás esetén a modul bizonyos függvényeit teszteljük. (Különben semmi sem történik.)

```
if __name__ == '__main__ ':
    welcome ( 'world ')
    welcome ( 'monde ' , 'Bonjour ')
    welcome ( 'world ' , mark ='... ')
    fib (100)
```

Az **examples.py** fájl programként történő végrehajtás a a következő eredményt adja. Az eredmény megegyezik, akár explicit módon használjuk az interpretert, akár nem.

```
$ python examples .py
Hello world !
Bonjour monde !
Hello world ...
1 1 2 3 5 8 13 21 34 55 89
\ begin { verbatim }
$ ls -l
total 100
-rwxr -x--- 1 rafi rafi 1046 2005 -03 -11 11:51 examples .py*
$ ./ examples .py
Hello world !
Bonjour monde !
Hello world ...
1 1 2 3 5 8 13 21 34 55 89
```

Megjegyzések Az import konstrukció alkalmazása esetén az interpreter a kért modult a **PYTHONPATH** környezeti változóval leírt elérési útvonalon keresi. Ez a változó alapértelmezetten azt a könyvtárat tartalmazza, ahová a Pythont telepítettük és az aktuális könyvtárat.

Amokor a **from** modul **import * ** konstrukciót alkalmazzuk, akkor a modul teljes tartalmát importáljuk, kivéve azokat a definíciókat, amiknek a neve „_” -vel kezdődik (ami a Pythonbeli private fogalmat tükrözi). Ennek az importálástípusnak az alkalmazása a kódírást könnyíti, ami interaktív módban érdekes. Minden esetre, ha megszűnik a modulnévvel mint prefixummal történő minősített hivatkozás, akkor felléphetnek a különböző modulokból importált funkciók (függvények, osztályok) közötti ütközések.

A modulok hierarchikus struktúrába rendezhetők. Ez esetben az almodulokat tartalmazó modulok, amiket package-eknek is neveznek, a filerendszer könyvtáraiként vannak definiálva. A modulok elnevezése tehát konkatenációval történik : **modul.almodul**. A kód portabilitásának egyszerűbbé tételéhez mindegyik könyvtárnak tartalmazni kell egy **__init__.py** nevű fájl-t. Ahhoz, hogy lehetséges legyen valamennyi almodul betöltése, ennek a fájl-nak tartalmazni kell a package moduljainak listáját az **__all__** változóban.

```
$ cat graphical / __init__ .py
__all__ = [ 'basic ' , 'advanced ']
```

2.1. Néhány standard és hasznos modul

A Python nagyszámú modult biztosít a felhasználónak. Most csak négy nélkülözhetetlennek tekintett modult mutatok be :

sys a futtató környezethez kapcsolódó paramétereket és függvényeket szolgáltatja,
string a karakterláncokon végezhető szokásos (a **string** osztály metódusaival ekvivalens) müveleteket szolgáltatja,
re a mintakereséshez és helyettesítéshez szolgáltatja a reguláris kifejezések támogatását
os az operációs rendszer generikus szolgáltatásaihoz nyújt hozzáférést.

A felsorolt modulok esetében beutatom és példákon illusztrálom az egyes modulok alapfunkcióit.

2.1.1. A sys modul

Néhány konstans a modulból :

`argv` a parancssorban átadott paraméter szekvencia (argv[0] a script nevét jelenti)
`stdin, stdout, stderr` A standard bemeneteket és kimeneteket reprezentáló filetípusú objektumok. Ezek az objektumok minden olyan objektummal helyettesíthetők, amiknek van write metódusa.
`path` A PYTHONPATH környezeti változóban tárolt elérési utakat tartalmazó szekvencia. Ez a szekvencia dinamikusan módosítható.

```
>>> sys. path . append ( '/ tmp/ python ')
['' , '/ usr/ lib / python2 .2 ' , '/usr /lib / python2 .2/ plat - linux2 ',
'/usr/lib/ python2 .2/ lib -tk ' , '/usr/lib/ python2 .2/ lib - dynload ',
'/usr/lib/ python2 .2/ site - packages ' , '/tmp/ python ']
```

`platform` Az operációs rendszer neve.

```
>>> sys. platform
'darwin '
```

`ps1, ps2` A promptok értékét — alapértelmezetten „>>>” és „...” — tartalmazó változók.

Néhány függvény a modulból :

exit([arg]) végetvet a programvégrehajtásnak, **arg** a kilépés státusa. Ez a függvény figyelembe veszi a **finally** klauzulában megadott „takarító” utasításokat (lásd 3.2.2.2 fejezetet).

2.1.2. A `string` modul

Ez a modul számos karakterlánc manipulálására szolgáló konstanst és függvényt tartalmaz. Általában ajánlott az egyenértékű **`string`** objektumok metódusainak a használata.

2.1.2.1 Néhány konstans

A string modul konstansai karakterkészleteket definiálnak :

```
>>> string . lowercase
' abcdefghijklmnopqrstuvwxyz '
>>> string . uppercase
' ABCDEFGHIJKLMNOPQRSTUVWXYZ '
>>> string . letters
' abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ '
>>> string . digits
'0123456789 '
>>> string . whitespace
'\t\n\ x0b\ x0c\r '
>>> string . punctuation
'!"#$ %&\ '()*+ , -./:; <= >?@ [\\]^_ `{|}~ '
```

2.1.2.2 Fontosabb függvények

`lower upper cap*` A kis-, nagybetűk kezelését teszik lehetővé egy karakterláncban : kisbetűssé, nagybetűssé, kapitálissá teszik a mondatot vagy a szavakat (a betűközök csökkentésével).

```
>>> string . lower ( 'FOO ')
'foo '
>>> string . upper ( 'foo ')
'FOO '
>>> string . capitalize ( 'foo ')
'Foo '
>>> string . capwords ( ' hello world ! ')
'Hello World !'
```

`strip expandtabs` Lehetővé teszi a betűközök kezelését egy karakterláncban a nem szignifikáns fehér betűközök törlésével vagy a tabulátorok fix-számú betűközzel való cseréjével.

```
>>> string . strip ( ' hello world ! \n ') # [2 nd arg]
'hello world !'
>>> string . expandtabs ( '\ thello world !' , 4)
' hello world !'
```

`find` részstring keresését teszi lehetővé egy karakterláncban (a karakterlánc elejétől vagy a végétől kezdve az **`rfind`** esetén) visszatérési értékként azt az indexet megadva, ahol a rész-stringet először megtalálta vagy -1 -et. (Az index függvény hasonló, csak egy kivételt generál abban az esetben, ha nem találta meg a rész-stringet a karakterláncban.)

```
>>> string . find ( 'bonjour le monde ' , 'on ')
1
>>> string . rfind ( 'bonjour le monde ' , 'on ')
12
```

```
>>> string . rfind ( 'bonjour le monde ' , 'om ')
-1
```

split lehetővé teszi egy karakterlánc szavakból álló listára való felbontását (alapértelmezetten a betűközöknél vagy a második argumentumként megadott stringnél).

join a split iverz művelete. Egy stringlistából és összekötő karakterből hozhatunk létre egy karakterláncot.

```
>>> string . split ( 'foo bar 123 ')
['foo ' , 'bar ' , '123 ']
>>> string . split (" hello world " , "wo")
['hello ' , 'rld ']
>>> string . join ([ 'foo ' , 'bar ' , '123 '], ';')
'foo;bar ;123 '
```

count replace Egy rész-string előfordulásának számlálását és egy másik stringgel való helyettesítését teszi lehetővé egy karakterláncban.

```
>>> string . count ( 'bonjour le monde ' , 'on ')

2
>>> string . replace ( 'bonjour le monde ' , 'on ' , 'ONON ')
'bONONjour le mONONde '
```

zfill center ljust rjust A kiíratás kezelését teszi lehetővé : a **zfill** adott helyiértéken ábrázolt számot balról nullákkal egészít ki, hogy minden helyiérték pozíció ki legyen töltve, a másik három függvény

```
 >>> string . zfill ( str (123) , 5)
'00123 '
>>> string . center ( 'hi!' , 10)
' hi ! '
>>> string . rjust ( 'hi!' , 10)
' hi!'
>>> string . ljust ( 'hi!' , 10)
'hi !
```

2.1.3. A `re` modul

Ennek a modulnak a segítségével manipulálhatunk reguláris kifejezéseket [2]. Alapértelmezetten ezek a kifejezések azonosak a Perl-es reguláris kifejezésekkel. *Jeffrey Friedl* « Mastering Regular Expressions » című műve jó kiegészítő azoknak, akiknek reguláris kifejezéseket kell alkalmazni. Hogy a kifejezésekben ne kelljen mindig a '\' -t használni, lehetőség van a kifejezést definiáló string **`'r' : r'(.*)\n'`** prefix-szel való ellátása révén *Raw regular expression* -ek használatára. Az utolsó kérdés, amit fel kell magunknak tenni mielőtt elkezdjük a reguláris kifejezések alkalmazását : a támogatott kódolási típus. A re modul az alábbi két modul valamelyikét használhatja :

- **`sre`** unicode támogatás (ez az alapértelmezetten használt modul, **`re`**-vel maszkolt implementáció)
- **`pre`** unicode támogatás nélkül (ennek a modulnak a használata nem tanácsolt)

2.1.3.1 Kereső műveletek

`search(minta, karakterlánc)` lehetővé teszi a minta megkeresését a karakterláncban. Visszatérési értékül egy **`SRE_Match`** típusú objektumot ad, ami a választ írja le vagy **`None`** -t. Az **`SRE_Match`** osztály két metódust bizosít, hogy megkapjuk a minta karakterláncbeli kező és végpozícióját (**`start()`** és **`end()`**).Itt a **`matcing`** metódus a **`span()`** -t használja. Ez utóbbi egy tuple-t ad, ami a két indexet tartalmazza.

```
>>> def matching ( res ):
... if res : print res. span ()
... else : print 'no matching '
>>> matching ( re. search ( 'ada ' , 'abracadabra '))
(5 , 8)
```

`match(minta, karakterlánc)` megvizsgálja, hogy a karakterlánc a mintával kezdődik-e. Visszatérési értékül egy **`SRE_Match`** típusú objektumot ad, ami a választ írja le vagy **`None`** -t.

```
>>> res = re. match ( 'abr ' , 'abracadabra ')
>>> matching ( res)
(0 , 3)
```

2.1.3.2 Manipulációs műveletek

`split(minta, karakterlánc)`egy minta alapján darabolja fel a karakterláncot. A következő példa szavakra darabol (minden szó, ami nem betűköz vagy írásjel).

```
>>> re. split ( '\W+' , 'Hello world ! ')
['Hello ' , 'world ' , '']
```

`sub(minta, helyettesítő, karakterlánc)` visszatérési értékként egy karakterláncot, amiben a minta összes előfordulását helyettesítette a második argumentumként megadott stringgel.

```
>>> re. sub ( 'hello ' , 'bonjour ' , 'hello foo hello bar ')
'bonjour foo bonjour bar '
```

2.1.3.3 Lefordított kifejezések használata

Amikor egy programban többször használunk egy reguláris kifejezést, akkor válik érdekessé a minta lefordítása. A funkciók az előzőekben látottakhoz (metódusok) hasonlóak, de a teljesítmény jobb.

`compile` egy lefordított reguláris kifejezést reprezentáló objektum létrehozását teszi lehetővé. Ha már létrehoztuk az objektumot, akkor a függvényeket ezekre az objektumokra hívjuk, nem pedig a modulra. Az **`SRE_Match`** objektum **`group()`** metódusa a **`search`** függvény által megtalált i -edik mintához (több mintából[16] alkotott kifejezés esetében, mint itt) tesz lehetővé hozzáférést.

```
>>> exp = re. compile ( '<a href ="(.*)" >(.*) </a>')
>>> res = exp. search ( 'Cliquer <a href =" foo. html ">ici </a >! ')
>>> matching ( res)
(8 , 34)
>>> res. group (0)
'<a href =" foo. html ">ici </a>'
>>> res. group (1)
'foo. html '
>>> res. group (2)
'ici '
>>> exp = re. compile (r'.*<a href ="(.*)" >.* </a >.* ')
>>> exp. sub (r'\1 ' , 'Cliquer <a href =" foo. html ">ici </a >! ')
'foo. html '
```

2.1.4. Az `os` modul

Ez a modul teszi lehetővé a hozzáférést az operációs rendszerhez, a rendszerprogramozást és a rendszer adminisztrációt. A funkciók (majdnem) minden rendszeren rendelkezésre állnak. Minden esetre mindig tudatában kell lennünk a kód portabilitási határainak : egy fájl-ra vonatkozó jogosultságoknak a fogalma nem minden operációs rendszeren hasonló.

Az **`os`** modul fő almoduljai a következők :

`path` hozzáférési utak manipulálása

`glob fnmatch` fájl-ok és könyvtárak mintával való

`time` hozzáférés az órához

`getpass` jelszavak, azonosítók és felhasználók manipulálása

2.1.4.1 Néhány konstans

`name` az os modul implementációjának nevét adja meg :**`posix, nt, dos, mac, java, etc.`**

`environ` a környezeti változókat tartalmazó szótár (Unix értelemben, de a Python interpreter indítására vonatkozóan is)

```
>>> os. environ
{'USER ': 'rafi ' , 'HOME ': '/ home / rafi ',
'PATH ': '/bin :/ usr/bin :/ opt/bin :/ home / rafi / bin ',
'HOSTNAME ': 'alfri . lifl .fr ',
```

16 Egy reguláris kifejezés belsejében zárójelek definiálnak egy mintát.

```
'PWD ': '/ home / rafi / enseign / python / scripts '}
```

2.1.5. Az os.path modul

Ez a modul a filenevek és fájl-ok manipulálására szolgáló (Unix alatt működő) függvényt.

`basename dirname` függvényekkel a elérési útból megadják a fájl illetve a könyvtár nevét.

```
>>> os. path . basename ( '/ tmp/ foo. txt ')
'foo.txt '
>>> os. path . dirname ( '/ tmp/ foo. txt ')
'/tmp '
```

`split join` lehetővé teszik egy elérési út feldarabolását, illetve konstruálását. Elérési utan konstruálására ajánlatos a join függvényt használni, mert az implicit módon figyelembe veszi az aktuális operációs rendszerre jellemző szeparátor karaktert.

```
>>> os. path . split ( '/ tmp/ foo. txt ')
('/tmp ' , 'foo .txt ')
>>> os. path . join ( '/ tmp ' , 'foo. txt ')
'/tmp/foo.txt '
```

`exists isdir isfile` függvényekkel tesztelhetjök egy adott típusú fájl (vagy könyvtár) létezését.

```
>>> os. path . exists ( '/ tmp/ bar. txt ')
False
>>> os. path . isdir ( '/ tmp ')
True
>>> os. path . isfile ( '/ tmp ')
False
```

2.1.6. A glob és fnmatch modulok

Ez a két modul a modulok nevével megegyező nevű függvényt szolgáltat.

`glob` a Unix **`ls`** parancsának a megfelelője. *Jocker*-ek alkalmazásával megadja egy könyvtárban lévő fájl-ok listáját.

```
>>> glob . glob ( '*. py ')
['hello .py ' , 'examples .py ']
>>> glob . glob ( '/ tmp /*. tmp ')
['/tmp / sv3e2 .tmp ' , '/tmp / sv001 .tmp ' , '/tmp / sv3e4 .tmp ']
```

`fnmatch és filter` lehetővé teszi fileneveket reprezentáló karakterláncokban mintával való egyezés tesztelését : az az, hogy egy filenév megfelel-e egy karakterlánc mintának, illetve egy lista mely elemei felelnek meg egy mintának.

```
>>> fnmatch . fnmatch ( 'examples .py ' , '*. py ')
1
>>> fnmatch . filter ([ 'examples .py ' , 'hello . pyc '], '*. py ')
['examples .py ']
```

2.1.7. A getpass modul

Ez a modul lehetövé teszi a felkapcsolódott felhasználó nevének és jelszavának a kérését a rendszertől. Az utóbbit „rejtve”.

getuser() kéri a rendszertől a felhasználó login nevét

```
>>> getpass . getuser ()
'rafi '
```

getpass() (a beírt karakterek maszkolásával) kéri a felhasználó jelszavát. Ez a függvényhívás mindaddig blokkolva van, amíg a felhasználó ben nem írt valamit.

```
>>> p = getpass . getpass () # bloquant jusqu 'au '\n'
Password :
>>> print p
' quelmauvaismotdepasse '
```

2.2. Beépített ffunkciók (built-in -ek) a Pythonban

A built-in-ek magába a Python interpreterbe kódolt funkciók és bizonyos értelemben maga az interpreter. Ezek a funkcionalitások nem Pythonban vannak megírva, mert ezeket nagymértékben használják és a C-ben történő implementálásuk révén jobb a teljesítményük.

2.2.1. A fájl-ok

2.2.1.1 A fileobjektumok

A fájl-okat **fájl** típusú objektumok reprezentálják. Szövegesek vagy binárisok lehetnek, írhatjuk vagy olvashatjuk őket.

open alapértelmezetten olvasásra megnyit egy fájl-t (létrehoz egy fileobjektumot).

```
>>> foo = open ( '/ tmp/ foo. txt ')
>>> foo
<open file '/tmp/foo.txt ' , mode 'r' at 0 x81a3fd8 >
```

close lezár egy fájl-t(de nem rombolja le az asszociált objektumot).

```
>>> foo. close ()
>>> foo
<closed file '/tmp/foo.txt ' , mode 'r' at 0 x81a3fd8 >
```

2.2.1.2 Olvasás fájl-ban

Többféle módja van a fájl-ból történő olvasásnak.

readline() egyszerre egy sort olvas a fájl-ból (az aktuális pozíciótól a következő \n -ig).

```
>>> foo = open ( '/ tmp/ foo. txt ' , 'r')
>>> print foo. readline ()
'hello world !\n'
```

readlines() lehetővé teszi egy fájl összes sorának egyszerre történő olvasását (egy stringekből álló szekvencia a visszatérési értéke).

```
>>> foo. readlines ()
['bonjour le monde !\n' , 'au revoir le monde !\n']
```

`read([n])` lehetővé teszi, hogy az aktuális pozíciótól kezdve olvassuk az egész fájl-t, vagy n byte-ot, ha megadjuk az argumentumot. A függvény visszatérési értéke egy string. A **`seek`** függvény az abszolút módon történő mozgást teszi lehetővé a fájl-ban : itt a fájl elejét adjuk meg (0 index).

```
>>> foo. seek (0) ; foo. read ()
'hello world !\ nbonjour le monde !\ nau revoir le monde !\n'
>>> foo. close ()
```

Az olvasásmódok összehasonlítása A fájl-ok olvasására nincs ideális megoldás. A helyzettől és az igényeinktől függően kell választanunk.

- Egy fájl globális olvasása (**`readlines()`**) az információ megszerzésének vonatkozásában hatékony, mert egy egyszeri diszkhezfordulást, majd pedig egy szekvenciának a memóriában történő bejárását jelenti. Minden esetre ez a megoldás memóriaigényes : képzeljünk el egy 500 Mbyte-os szövegfile-t a memóriában.
- A soronként történő olvasás több olvasási műveletet igényel, mert nagyszámú diszkhezfordulást jelent, hogy kisebb információ adagokhoz jussunk. Minden esetre ez a megközelítés nagyméretű fájl-ok manipulálását teszi lehetővé : egy 10 Gbyte-os fájl olvasható egy 64 Mbyte-os PC-vel.

2.2.1.3 Írás fájl-ba

Fájl-ba történő íráshoz a fájl-t írásra meg kell nyitni. A fájl megnyitásakor tehát megadunk egy második paramétert '**w**' -t (lásd a C nyelv filemegnyitási módjait, bináris fájl-ok esetén egy '**b**' -t is). Amíg nincs lezárva a fájl, addig nem garantált, hogy a tartalma diszken van.

`write` egy vagy több szövegsort reprezentáló adatot írhatunk vele ('**`\n`**' használata).

```
>>> foo = open ( '/ tmp/ foo. txt ' , 'w')
>>> foo. write ( 'hello world !\n')
```

`writelines` egy (karakterlácokból álló) szekvenciában lévő adatokat ír ki. Ha a szekvenciában lévő mindegyik string egy szövegsort reprezentál a fájl-ban, akkor mindegyik stringnek tartalmazni kell a '**`\n`**' sorvége szekvenciát.

```
>>> lines = [ 'bonjour le monde !\n' , 'au revoir le monde !\n']
>>> foo. writelines ( lines )
>>> foo. close ()
```

2.2.2. Típuskonverziók

Típusnevek használata stringek, egészek, lebegő pontos számok, stb. átalakítására. A str függvénnyel minden objektumot karakterláccá alakíthatunk.

```
>>> str (123)
'123 '
>>> int ( '123 ')
123
>>> float ( '123 ')
123.0
>>> float (123)
123.0
>>> long ( '123 ')
```

```
123L
```

2.2.3. Dinamikus kiértékelés

A Python lehetővé teszi parancsok futás közbeni kiértékelését : végrehajt egy parancsot reprezentáló karakterláncot. A karakterlánc akár explicit módon le is lehet fordítva a végrehajása előtt.

`compile()` A paraméterként megadott kifejezés (vagy fájl) lefordított verzióját adja visszatérési értékül. A második argumentum határozza meg az error output helyét abban az esetben, amikor egy kivétel (exception) generálódik. Az utolsó argumentum határozza meg, hogy hogyan akarjuk a lefordított kifejezést használni, most az eval() függvénnyel akarjuk használni.

```
>>> obj = compile ( 'x + 1 ' , '<string >' , 'eval ')
>>> obj
<code object ? at 0 x81685d8 , file "<string >" , line -1>
```

`eval()` kiértékel egy paraméterként megadott kifejezést (akár le van fordítva, akár nincs lefordítva)

```
>>> x = 2
>>> eval ( 'x + 1 ')
3
>>> eval ( obj)
3
```

2.2.4. Assertions

Les assertions permettent de traiter les situations sans appel : soit la condition est

respectÂ´ee, soit le programme est arrˆetÂ´e. (Dans la pratique, une exception est levÂ´ee.)

`assert` kiértékel egy logikai kifejezést és egy hibaüzenetet (ami opcionális és vesszővel elválasztva adunk meg) kiírva leállítja a programot, ha a logikai kifejezés értéke hamis (lásd az 1.3.2 fejezetet). Akifejezésben lehetnek függvényhívások.

```
>>> assert 1
>>> assert 0 , 'oups '
Traceback ( most recent call last ):
    File "<stdin >" , line 1 , in ?
AssertionError : oups
>>> try : assert 0
    except AssertionError : print 'sic '
...
sic
```

2.3. Gyakorlatok

2.3.1. A save utility

Felhasználjuk az **`os.stat`** függvényt, ami lehetővé teszi, hogy megtudjuk a fájl-ok módosításának a dátumát, egy fastruktúra másoló segédprogram megírását. Ennek a segédprogramnak (az első alkalmat kivéve) inkrementálisan kell a módosított fájl-ok másolatát elkészíteni.

```
>>> print os. stat . __doc__
```

```
stat ( path ) - > ( st_mode , st_ino , st_dev , st_nlink ,
  st_uid , st_gid , st_size , st_atime , st_mtime , st_ctime )
Perform a stat system call on the given path .
```

2.3.2. Adatok kinyerése

Ez a gyakorlat a reguláris kifejezések *(regular exressions)* modul függvényeinek a használatát javasolja adatoknak egy szövegfile-ból történő kinyerésére. Írjunk egy mailbox típusú szövegfileelemző programot, ami az e-mailt küldő valamennyi személy listáját elemzi, a listát névsorba rendezi a feladó címe szerint a frissebb e-mail-ekkel kezdve.

3. Elmerülünk az objektumokba

3.1. Objektumok, mint objektumok

3.1.1. Az objektum orientált programozás alapelveinek ismétlése

Példányosítás (instantiation) Egy objektumot egy minta, - az osztálya - alapján hozunk létre, ami definiál egy struktúrát (az attribútumokat) és viselkedést (a metódusokat). Egy osztály egy objektumtípust definiál. Egy objektum egy egyedi osztálynak egy példánya.

Egységbezárás (encapsulation) Az adatok «el vannak rejtve» az objektumok belsejében, az adatokhoz való hozzáférét metódusokkal kontroláljuk. Egy objektum állapotát nem szabad közvetlen módon manipulálni. Az objektumra úgy tekinthetünk, mint egy szolgáltatóra, nem pedig mint adatokra.

Polimorfizmus Az azonos interface-ű (a Python esetében elegendő a metódusok szignatúrája) objektumokat általánosan módon manipulálhatjuk, még akkor is, ha az egzakt típusuk eltérő. Ez az elv lehetővé teszi egy példánynak egy másikkal való helyettesítését is (amíg az interface-ek kompatibilisek).

Öröklés (inheritance) Ez a mechanizmus teszi lehetővé az alapdefiníciók, az alapértelmezett viselkedések újrafelhasználását és bizonyos viselkedési formák specializálását. Két osztály közötti öröklési viszony nem korlátozódik egyszerűen a kód megspórolására. Annak az osztálynak, amelyik örököl, annak egy olyan osztálynak kell lenni, mint amitől örököl : a macska egy állat, tehát a macska az állat osztályegy alosztálya.

3.1.2. Objektumok és hivatkozások

A Pythonban a világ homogén.

Minden objektum : a stringek, az egészek, a listák, az osztályok, a modulok, stb. Dinamikusan minden manipulálható.

Minden manipulálható hivatkozással : egy változó egy hivatkozást tartalmaz egy objektumra, egy objektumra több változó is hivatkozhat.

Egy függvény, egy osztály, egy modul hierarchikusan szervezett névterek : egy modul osztályokat tartalmaz, amik függvényeket tartalmaznak.

3.1.3. Osztályok

3.1.3.1 Definiálás és példányosítás

Egy osztály definiálása a `class` kulcsszóval a struktúrálási szabálynak (lásd 1.3.1 fejezetet) megfelelően történik. Minden metódus függvényként van definiálva, melynek első argumentuma (a `self`) azt az objektumot reprezentálja, melyre a metódus a végrehajtásakor alkalmazva lesz. A Java-ban és a C++ -ban a `this` implicit módon van definiálva, a Pythonban explicit és mindig első paramétere a metódusnak. A nevét szabadon választhatjuk, általában a `self` -et szoktuk választani. Egy metódusnak, aminek nem adunk át paramétert, ezért egy argumentuma mégis lesz.

```
>>> class Dummy :
        def hello ( self ):
            print 'hello world !'
```

A példányosításnál (objektum generálásnál) nem alkalmazunk semmilyen speciális kulcsszót sem (a Pythonban nincs **new**). Elegendő, ha az osztály nevét egy zárójel követi, ami vagy tartalmaz paramétereket vagy sem. A metódushívás a pont-operátoros minősített névmegadással történik. A metódust egy objektumra hivatkozó változón hívjuk.

```
>>> d = Dummy ()
>>> d
<__main__ . Dummy instance at 0 x817b124 >
>>> d. hello ()
hello world !
```

3.1.3.2 Az attribútumok és a constructor definiálása

Az attribútumok az első értékadásukkor vannak definiálva. Egy változót akkor tekintünk attribútumnak, ha az objektumhoz van kapcsolva : megelőzi a **self**. Az objektum egy attribútumához vagy metódusához való minden hozzáférés kötelezően a **self** -en keresztül történik. A constructor az **__init__** -nek nevezett metódus. Mint minden metódus esetében, úgy a constructor esetében is egyes paramétereknek lehetséges alapértelmezett értékük.

```
>>> class Compteur :
        def __init__ ( self , v = 0):
            self .val = v
        def value ( self ):
            return self .val
```

Software mérnöki értelemben mindig ajánlatos az objektumokállapotát (az attribútumait) inicializálni a példányosításukkor, tehát a constructorban. Másrészt az olyan constructor, ami semmit se csinál, egyáltalán nem hülyeség! Ez azt jelenti, hogy a constructor létezik : a constructor hiánya nem feledékenység lesz.

3.1.4. Öröklés

A Python támogatja az egyszeres és a többszörös öröklést is. Egy öröklési viszonyban meg kell adni a szülőosztály nevét (az osztály definiálásakor az osztály neve után következő zárójelben) **és** explicit módon kell hívni a szülőosztály (super-class) constructor-át.

```
class A:
    def __init__ ( self , n = 'none '):
        self . _name = n
    def name ( self ):
        return self . _name

class B (A):
    def __init__ ( self , val = 0 , n = 'none '):
        A. __init__ ( self , n)
        self . _wheels = val
    def wheels ( self ):
        return self . _wheels
```

Többszörös öröklés esetén elég megadni a szülőosztályakat és hívni kell a megfelelő constructor-okat.

```
class C:
    def __init__ ( self , t = ''):
        self . _title = t
    def title ( self ):
        return self . _title

class D (A , C):
    def __init__ ( self , n = 'none ' , t = ''):
        A. __init__ ( self , n)
        C. __init__ ( self , t)
    def fullname ( self ):
        return self . _title + ' ' + self . _name
```

A következő példa az előző három osztályt használja. Semmi különlegeset nem tapasztalunk, a használatkor semmit sem változtat az ha az osztályt teljes egészében definiáltuk, vagy pedig örökléssel hoztuk létre.

```
>>> a = A ( 'rafi ')
>>> print a. name ()
rafi
>>> b = B (4 , 'car ')
>>> print b. name ()
car
>>> print b. wheels ()
4
>>> d = D (t = 'dr ')
>>> print d. fullname ()
dr none
```

A problémás többszörös öröklés A többszörös öröklés akkor okoz problémát, ha két örökléssel létrehozott osztálynak azonos nevű metódusa van. Ebben az esetben a névütközés feloldására a metódusokat az őket definiáló osztály nevével, mint prefixummal adhatjuk meg, vagy alkalmazhatunk metódus-aliasokat.

```
class X:
    def name ( self ):
        return 'I am an X'
class Y:
    def name ( self ):
        return 'I am an Y'
class Z (X , Y):
    xname = X. name
    yname = Y. name
    def name ( self ):
        return 'I am an Z , ie ' + self . xname () + \
               ' and ' + self . yname ()
```

A következő példa az előző X, Y, Z osztályok alkalmazását mutatja be úgy, hogy a **name()** metódust hívja a három generált objektumon.

```
>>> for clss in [X , Y , Z]:
        obj = clss ()
        print obj. name ()
...
I am an X
I am an Y
I am an Z , ie I am an X and I am an Y
```

3.1.5. Egyszerű önelemzés (introspectio)

A **dir()** függvény bármelyik objektumnak megadja a tartalmát (az attribútumainak és a metódusainak a listáját). A **__doc__** attribútum-mal az előbbi mindegyik objektumnak az alapdokumentációját szolgáltatja. Ezért érdemes jól dokumentálnunk a kódot (autodokumentáció).

```
>>> dir ( Compteur )
['__doc__ ' , '__init__ ' , '__module__ ' , 'value ']
>>> c = Compteur ()
>>> dir (c)
['__doc__ ' , '__init__ ' , '__module__ ' , 'val ' , 'value ']
```

A type() függvény megadja egy hivatkozás típusát.

```
>>> type (c)
<type 'instance '>
>>> type ([1 , 2])
<type 'list '>
```

3.1.6. Osztályok és attribútumok

Egy osztály csak attribútumokat tartalmaz. Egy egyszerű szótárnak tekinthetjük, ami név/hivatkozás asszociációkat tartalmaz. Az osztály egy függvénye (vagy metódusa) valójában egy végrehajtható attribútum (*callable*).

```
>>> class OneTwoThree :
        value = 123 # reference un entier
        def function ( self ): # reference une fonction
        return self . value
...
>>> ott = OneTwoThree ()
>>> dir ( ott )
['__doc__ ' , '__module__ ' , 'function ' , 'value ']
>>> ott . function
<bound method OneTwoThree . function of < __main__ . OneTwoThree
```

Egy attribútum és egy metódus közötti névütközés esetén az attribútumé az elsőbbség. Itt az interpreter megmagyarázza, hogy a name attribútumot nem használhatjuk úgy, mintha függvény (vagy metódus) lenne, mert az egy karakterlác, tehát nem kérhetjük a végrehajtását (*object is not callable*).

```
>>> class Conflict :
        def __init__ ( self ):
            self . name = 'Conflict '
        def name ( self ):
            return 'You will never get this string !'
...
>>> c = Conflict ()
>>> c. name ()
Traceback ( most recent call last ):
   File "<stdin >" , line 1 , in ?
TypeError : 'str ' object is not callable
```

Megállapodás szerint '_'-prefixummal lájuk el egy osztály azon alkotóit, amiket nem akarunk publikusnak tekinteni. Az egységbezárás elve alapján az attribútumoknak mindig védettnek kell lenni. Az igaz, hogy a Pythonban a « protected » fogalma a bizalmon alapul, de 180 km/h -val hajt a városban ki a felelős az autó vagy Ön ?

```
>>> class NoConflict :
        def __init__ ( self ):
            self . _name = 'NoConflict '
            def name ( self ):
            return self . _name
...
>>> c = NoConflict ()
>>> print c. name ()
NoConflict
```

Lehetőség van rá, hogy dinamikusan definiáljunk attribútumokat egy osztálynak vagy egy objektumnak. Ez a fajta adottság nem korlátozódik a játékos vonatkozásokra, néha ez a legjobb megoldás egy problémára. Minden esetre ügyeljünk rá, hogy mit csinálunk amikor dinamikusan módosítjuk az objektumokat.

```
>>> class Empty :
        pass
>>> Empty . value = 0
>>> def funct ( self ):
        self . value += 1
        return self . value
...
>>> Empty . funct = funct
>>> dir ( Empty )
['__doc__ ' , '__module__ ' , 'funct ' , 'value ']
>>> e = Empty ()
>>> e. funct ()
1
```

Néhány ismert attribútum

__doc__ az osztály, az objektum, a modul, a függvény, stb. dokumentációját tartalmazza.
__module__ az osztály, az objektum, a modul, a függvény, stb. definícióját tartamazó modul nevét tárolja
__name__ a függvény vagy a metódus nevét tartalmazza
__file__ a modul kódját tartalmazó fájl nevét tárolja

3.1.7. Egy kis reflexió

A reflexiós technikák lehetővé teszik egy objektom attribútumainak (és metódusainak) kiderítését (introspection), valamint dinamikus és automatikus manipulációját. Ezek a technikák például olyan dinamikus kódváltoztatásokra használhatók, mint amilyen egy plug-in mechanizmus kivitelezése. Alkalmazásuk szokásos az olyan fejlesztő környezetekben, melyek automatikus kódkiegészítést végeznek (bármilyen információ keresésére van szükség). A Pythonban három alapfüggvény van :

`hasattr()` megvizsgálja, hogy létezik-e egy attribútum

`getattr()` egy attribútum értékéhez fér hozzá

`setattr()` beállítja egy attribútum értékét (létrehozza az attribútumot, ha nem létezik).

A következő példában az **`Empty`** osztály kiterjesztett objektumát fogjuk manipulálni : miután ellenőriztük a létezését, dinamikusan kinyerjük belőle a **`funct`** nevű metódusát, hogy azután a végrehajtását kérjük. Végül egy **`name`** nevű új attribútumot definiálunk, aminek a **`myempty`** értéket állítjuk be.

```
>>> hasattr (e , 'funct ')
1
>>> f = getattr (e , 'funct ')
>>> f
<function funct at 0 x81b7334 >
>>> f (e)
2
>>> setattr (e , 'name ' , 'myempty ')
>>> e. name
'myempty '
```

3.1.8. Egy kicsit több reflexió az `inspect`-tel

Az inspect modul kiegészítő eszközöket ad az önelemzéshez (introspectio), például annak kiderítéséhez, hogy mit tartalmaz egy osztály, egy objektum vagy egy modul. A következő példában kinyerjük az **`E`** osztály attribútumait, majd pedig az **`E`** osztály egy objektumának attribútumait nyerjük ki. Az objektum (osztály vagy példány) minden egyes attribútuma számára egy-egy tuple van megadva, amik az attribútum nevét és értékét tartalmazzák. Megállapíthatjuk, hogy a különbség az **`f`** metódus állapotára korlátozódik : nem kapcsolt az osztály esetében, tehát közvetlenül nem hajtható végre; illetve a példányhoz kapcsolódik az objektum esetében, amit egy **`getattr`** után közvetlenül végrehajtathatunk.

```
>>> class E:
        def f ( self ):
            return 'hello '
...
>>> e = E ()
>>> import inspect
>>> inspect . getmembers (E)
[( '__doc__ ' , None ) , ( '__module__ ' , '__main__ '),
 ('f' , < unbound method E.f >)]
>>> inspect . getmembers (e)
[( '__doc__ ' , None ) , ( '__module__ ' , '__main__ '),
 ('f' , < bound method E.f of < __main__ .E instance at 0 x825ec54 > >)]
```

Ez a modul azt is lehetővé teszi, hogy megtudjuk, éppen mit manipulálunk : mi az objektum (osztály, példány, attribútum) típusa. Az **ismethod()** függvénnyel tudhatjuk meg, hogy egy adott objektum (kapcsolt vagy nem kapcsolt) metódus e.

```
>>> inspect . isclass (E)
1
>>> f = getattr (e , 'f')
>>> inspect . isfunction (f)
0
>>> inspect . ismethod (f)
1
>>> F = getattr (E , 'f')
>>> inspect . ismethod (F)
1
```

Az alap introspectio és az **inspect** modul összekapcsolása lehetővé teszi egy objektum metódusainak automatikus alkalmazását : egy dinamikusan kinyert metódus általános hívását azután, hogy ellenőriztük, valóban egy metódusról van-e szó. A következő példa is bemutatja a metódushívás kétféle módját és a **self** jelentését : az interpreter lefordítja az **a.foo()** -t **A.foo(a)** -ra (úgy tekinti, hogy **a** a **A** osztály egy objektuma).

```
>>> f1 = getattr (e , 'f')
>>> f2 = getattr (E , 'f')
>>> if inspect . ismethod ( f1 ):
        f1 () # 'f1 ' est liee a 'e'
...
'hello '
>>> if inspect . ismethod ( f2 ):
        f2 (e) # 'f2 ' n'est pas liee , argument 1 == self
...
'hello '
```

Sőt az **inspect** modul lehetővé teszi egy objektum forráskódjához való dinamikus hozzáférést. Minden esetre ez nem áll fenn az interaktív módon beírt kódra (ce qui est somme toute normale vu qu'il n'est pas stockÂ´e dans un fichier).

```
>>> from examples import Conflict
```

getfile() megadja az objektumot definiáló fájl nevét (TypeError, ha ez a művelet nem lehetséges)

```
>>> inspect . getfile ( Conflict )
'examples .py '
```

getmodule() megadja az objektumot definiáló modul nevét (erre nincs garancia például dinamikusan létrehozott kód esetén).

```
>>> inspect . getmodule ( Conflict )
<module 'examples ' from 'examples .py '>
```

getdoc() az objektum dokumentációját adja visszatérési értéknek

```
>>> inspect . getdoc ( Conflict )
' Illustration de conflit attribut / methode '
```

getcomments() az objektum definícióját megelőző kommentsort adja visszatérési értéknek

getsourcelines() visszatérési értéknek egy tuple-t ad, ami a paraméterként megadott objektumot definiáló forráskódsorok listáját és a definíció filebeli kezdő sorát tartalmazza.

```
>>> lines , num = inspect . getsourcelines ( Conflict )
>>> for l in lines : print num , l , ; num += 1

34 class Conflict :
35     ''' Illustration de conflit attribut / methode '''
36     def __init__ ( self ):
37         self . name = 'Conflict '
38     def name ( self ):
39         return 'You will never get this string !'
```

getsource() hasonlít az előzőhöz, de csak a definíció kódját adja visszatérési értékül egy karakterlánc formájában.

3.1.1. Osztályok kontra modulok

Úgy tűnik, a Python-fejlesztőknek el kell fogadni a következő víziót:

- Osztályokat kell használni, amikor
  - az adatoknak egy állapotot kell reprezentálni és védetteknek kell lenni
  - az adatok és az adatok kezelése közötti kapcsolat szoros
- Modulokat és függvényeket kell használni, amikor
  - az adatok és az adatok kezelése« független » (pl.: data mining[17])
  - az adatok adatbázisokban vagy fájl-okban vannak tárolva és nincsennek módosítva

3.2. Kivételek (exceptions) a Pythonban

3.2.1. Definíció és bevezetés

A kivételek (exceptions) olyan objektumok (vagy majdnem), mint a többi, vagyis osztályok definiálják őket, mint a többi objektumot. Hajlamosak rá, hogy semmit sem csináljanak (a kivétel-osztályok ritkán tartalmaznak műveleteket). Egy kivétel-osztály minimális definíciója az **Exception** alaposztály kiterjesztéséből áll.

```
>>> class MonErreur ( Exception ):
... pass
...
```

A **raise** kulcsszóval generálhatunk egy kivételt (ami standard kivétel vagy felhasználó által definiált kivitel lehet). Ha információt akarunk asszociálni egy kivitelhez, elég ha a kivitel típusa után írjuk az üzenetet (egy vesszővel elválasztva)[18].

17 Hasznos adatok kinyerése nagyméretű adathalmazokból vagy adatbázisokból

18 A kivételeket úgy generáljuk, mint az objektumokat. Az interpreterben egy osztály egy objektumként van definiálva. Tehát az osztály definícióját kivételként generálhatjuk. Ez nem olyan sokkoló az egyszerű esetekben (kivétel hibaüzenettel).

```
>>> raise MonErreur
Traceback ( most recent call last ):
  File "<stdin >" , line 1 , in ?
__main__ . MonErreur
>>> raise MonErreur ()
Traceback ( most recent call last ):
  File "<stdin >" , line 1 , in ?
__main__ . MonErreur
>>> raise NameError , 'cela coince '
Traceback ( most recent call last ):
  File "<stdin >" , line 1 , in ?
NameError : cela coince
```

Egy kivétel tartalmazhat attribútumokat és metódusokat is (ezek általában információ szolgáltatására valók). Itt a standard **__str__** metódust (az objektum strigre fordítását) használjuk arra, hogy egyszerűen hozzáférjünk a kivételben lévő értékhez (alapértelmezetten a kivétel mechanizmus ezt a metódust hívja). Ahhoz, hogy a kivételt információval lássuk el, elegendő a **raise** utasítás alkalmazásakor létrejövő objektum constructorának paramétereket átadni.

```
>>> class MonErreurToo ( Exception ):
        def __init__ ( self , val = None ):
            self . _value = val
        def __str__ ( self ):
            return str ( self . _value )
...
>>> raise MonErreurToo (12)
Traceback ( most recent call last ):
File "<stdin >" , line 1 , in ?
__main__ . MonErreurToo : 12
```

3.2.2. Kivételek kezelése

A kivételek kezelésének két megközelítése létezik a Pythonban. Az igényeink alapján választunk közülük:

- mit kell csinálni probléma esetén
- que faut il toujours faire méme s'il y a un probléme.

3.2.2.1 Try ... except

A **try … except** konstrukció kivétel generálódása esetén lehetővé teszi a probléma kezelésének megkísérlését. Az **else** klauzula opcionális és abban az esetben, ha minden rendben zajlott, lehetővé teszi múveletek végzését. A konstrukciónak több **except** klauzulája lehet a különböző kivételes esetek kezelésére.

```
try:
    <utasítások >
( except < kivétel >:
    <kivételkezelő utasítások >)+
[ else
    <utasítások a korrekt programvégrehajtás esetére>]
```

Egy <kivétel> azonosításának többféle formája lehetséges:

- **`ExceptionType`** akkor hasznos, amikor a kivételnek csak a típusa fontos, például azért, mert a kivétel nem tartalmaz információt
- **`ExceptionType, változó`** lehetővé teszi a megadott típusú kivétel kinyerését a változóba és a kivétel adatainak és műveleteinek a felhasználását
- **`(ExceptionType1, ExceptionType2, …)`** lehetővé teszi kivételek egy csoportjának a számításba vételét, mert a speciális eset kezelése dswd

1. példa: Szövegfile olvasása : csak akkor olvasuk a fájl-t, ha rendben megnyitottuk. Az **`else`** klauzulát annak a kódnak tartjuk fenn, amit csak abban az esetben kell végrehajtani, ha nem generálódott kivétel. Ez lehetővé teszi a potenciálisan problémát okozó függvényhívás izolálását a **`try`** blokkban (nem tesszük az összes kódot ebbe a blokkba anélkül, hogy tudnánk valójában mi okozta a problémát).

```
>>> try:
        f = open ( '/ tmp/ bar. txt ')
    except IOError , e:
        print e
    else :
        for line in f. readlines ():
            print line ,
        f. close ()
...
hello world !
bonjour le monde !
```

2. Több kivétel kezelése : a kiíratás tartalma a generált kivételtől fog függeni (a kivétel véletlenszerűen generálódik). Mivel egy karakterlánc egy objektum, ezért kivételként használható. *Cette mani`ere de faire est toutefois `a proscrire.*

```
>>> try:
        if int ( random . random () * 100) % 2:
            raise 'impair '
        else :
            raise 'pair '
    except 'pair ':
        print 'c'est un nombre pair '
    except 'impair ':
        print 'c'est un nombre pair '
...
c'est un nombre pair
```

3.2.2.2 try … finally

Bármi is történik, a **`try … final`** konstrukció lehetővé teszi a helyzet kezelését még akkor is, ha egy kivétel (exception) generálódott. A **`finally`** blokk kiértékelésére sor kerül egy **`try`** blokkbeli **`return`** előtt függetlenül attól, hogy generálódott-e kivétel vagy sem. Figyelem : egy **`finally`** blokkbeli **`return`** kimaszkol egy **`try`** blokkbeli **`return`** -t.

```
try:
    < utasítások >
finally :
    < műveletek >
```

Figyelem : ebben az esetben nem tudjuk, hogy mi volt hibás. Nincs hibakezelő eszközünk a finally blokkban, ami lehetővé tenné a kivételek« kiszűrését ».

```
>>> try :
          raise KeyboardInterrupt
    finally :
          print 'ki C ki A ta P <ctrl -C > ? '
...
ki C ki A ta P <ctrl -C > ?
Traceback ( most recent call last ):
File "<stdin >" , line 2 , in ?
KeyboardInterrupt
```

Nem lehet a két formát egy blokkban kombinálni, tehét néha szükség van a két forma egymásba ágyazott használatára.

3.2.3. Kivételek kezelése és az öröklés

Gyakori probléma : ügyeljünk az except -ek sorrendjére! Abban az esetben, amikor a kivételeket öröklési relációval definiáljuk, mindig a legspecifikusabb kivételek kezelésével kell kezdenünk. Ellenkező esetben a legáltalánosabb kivételek kezelése maszkolja a legspecifikusabb kivételek kezelését, amiket soha sem fogunk kezelni.

```
class A ( Exception ): pass
class B (A): pass
>>> for x in [A , B]:
        try : raise x
        except B: print 'B',
        except A: print 'A',
A B
>>> for x in [A , B]:
        try : raise x
        except A: print 'A',
        except B: print 'B',
A A
```

3.3. Gyakorlatok

3.3.1. Az első osztályok

– Implementáljuk a Pile és Fájl osztályokat a listák, mint belső adatszerkezetek alkalmazásával és a következő interface figyelembe vételével (alkalmazzuk az öröklést).

```
class Base :
    def pop ( self ):
        pass
    def push ( self , elt ):
        pass
```

– Csináljuk meg a 2.4.2 gyakorlatot úgy, hogy a funkciókat egy osztályban implementáljuk!

3.3.2. Az állapot design pattern

4. A Python és az XML

4.1. XML DOM vízióval

4.1.1.

- Kezdetben volt az SGML
  - *Standard Generalized Markup Language*
  -
  - XML -lel piaci termékké alakítva
- A Python mindet támogatja
  - HTML, XHTML és SGML a HTML számára
  - XML, DOM és SAX

4.1.2. XML, milyen eszközök ?

Az XML dokumentumok Pythonnal való kezeléséhez több könyvtár is rendelkezésünkre áll. Ez a fejezet a standard **`xml`** package egy részére korlátozódik, aminek több subpackage-e van :

– **`dom`** : a DOM Pythonbeli implementációja, a minidom-ot tartalmazza,

– **`sax`** : a rendelkezésre álló implementációk egy szótára,

– **`parsers`** : a DOM és a SAX által belsőleg használt szövegelemzőt tartalmazza.

Kiegészítésként számos könyvtár és kiterjesztés áll rendelkezésünkre. Ezek például valamilyen adott alkalmazáshoz a legmagasabb szintű absztrakciókat és eszközöket szolgáltatják.

4.1.3. DOM, rövid ismétlés

Az XML fájl-ok kezelésére kizárólag a DOM-ot (**Document Object Model)** fogjuk használni. A DOM azt ajánlja, hogy egy XML dokumentumot a memóriában úgy kezeljünk, mint a dokumentum csomópontjait (nódusait) reprezentáló objektumfát. A standard DOM interface-ek a következők :

Interface	Mit reprezentál
Csomópont (Node)	A csomópontok alapinterface-e
csomópontlista (NodeList)	Csomópontok szekvenciája
Document	Egy teljes dokumentumot
Element	A hierarchia elemét
Attr	Egy csomópont attribútumát reprezentáló csomópont
Comment	Egy kommentet reprezentáló csomópont
Text	Szöveges adatok csomópontja

4.1.4. Egy fejezet példája

A következő lista egy XML példadokumentum, amit a továbbiakban az ilyen dokumentumok manipulálásának illusztrálására fogok felhasználni.

```
<?xml version ="1.0" ?>
<contacts >
   <contact name ="doe" firstname =" john ">
      <address >
        <road value ="10 , binary street " />
        <postal value =" 0001 " />
        <city value ="cpu" />
      </ address >
       <programming lang ="asm" />
   </ contact >
   <contact name =" dupont " firstname =" jean ">
      <address >
        <road value =" impasse de l'assembleur " />
        <postal value =" 0100 " />
        <city value =" dram " />
      </ address >
       <programming lang ="c" />
   </ contact >
   <contact name =" terprette " firstname =" quentin ">
      <address >
        <road value =" avenue du script " />
        <postal value =" 1001 " />
        <city value =" salt snake city " />
      </ address >
       <programming lang =" python " />
   </ contact >
</ contacts >
```

4.2. Navigálás egy DOM fában

4.2.1. A minidom „megteszi a maximumot”

A minidom modul a DOM implementálása a Pythonban. Az összes DOM alapinterface-t szolgáltatja és egy XML fájl- (illetve karakterlánc) elemzőt. Standardként rendelkezésre áll a környezetben (PyXML).

4.2.2. Egy XML dokumentum elemzése

A minidom.parse függvény lehetővé teszi egy XML fájl elemzését és a megfelelő DOM fa előállítását. Visszatérési értékként egy **`Document`** típusú objektumot ad.

```
>>> from xml .dom import minidom
>>> doc = minidom.parse( '/tmp/ contacts .xml ')
>>> doc
<xml.dom.minidom.Document instance at 0 x827fb9c >
```

4.2.3. Egy DOM fa bejárása

Számos művelet és attribútum vonatkozik egy DOM fa minden elemére, melyek lehetővé teszik a DOM fa bejárását.

`hasChildNodes()` jelzi, hogy egy csomópontnak gyermekcsomópontja van.

```
>>> doc.hasChildNodes()
1
```

`childNodes` hozzáférhetünk egy csomópont gyermekcsomópontjához (egy lista formájában).

```
>>> doc.childNodes
[<DOM Element : contacts at 137763676 >]
>>> doc.childNodes[0].hasChildNodes
1
```

`documentElement` megadja egy DOM dokumentum gyökérelemét.

```
>>> root = doc.DocumentElement
```

`firstChild, lastChild` egy csomópont első, illetve utolsó gyermekcsomópontjához ad hozzáférést.

```
>>> root.childNodes
[<DOM Text node "\n " >, <DOM Element : contact at 137757692 > ,
 <DOM Text node "\n " >, < DOM Element : contact at 137437676 > ,
 <DOM Text node "\n " >, < DOM Element : contact at 136723316 > ,
 <DOM Text node "\n" >]
>>> root . firstChild
<DOM Text node "\n ">
>>> root.lastChild
<DOM Text node "\n">
```

nextSibling, previousSibling a következő, illetve előző gyermekcsomópontot adja visszatérési értékül (illetve **None** -t, ha nincs több gyermekcsomópont). A számolás egy közös gyökérhez viszonyítva történik.

```
>>> current = root.firstChild
>>> while current :
        print current ,
        current = current.nextSibling
...
<DOM Text node "
" > <DOM Element : contact at 137757692 > < DOM Text node "
" > <DOM Element : contact at 137437676 > < DOM Text node "
" > <DOM Element : contact at 136723316 > < DOM Text node "
">
```

parentNode egy csomópont szülőcsomópontjához férhrtünk vele hozzá

```
>>> root
<DOM Element : contacts at 137763676 >
>>> root.firstChild
<DOM Text node "\n ">
>>> root.firstChild.parentNode
<DOM Element : contacts at 137763676 >
```

isSameNode() két csomópont egyenlőséget vizsgálja (az egyenlőséget az azonosság értelmében használom).

```
>>> root.firstChild.isSameNode( root.lastChild )
0
>>> root.firstChild.isSameNode( root.childNodes[0])
1
```

4.2.4. Keresés egy DOM fában

Egy DOM fában a keresés elsősorban a tag neve (a csomópont neve) szerint történik. A **Document** és **Element** típusú csomópontok esetében a **getElementsByTagName()** függvény megadja a megadott nevű tag gyermekcsomópontját (és az utóbbiak gyermekcsomópontjait).

```
>>> root.getElementsByTagName( 'contact ')
[<DOM Element : contact at 137757692 > ,
 <DOM Element : contact at 137437676 > ,
 <DOM Element : contact at 136723316 >]
>>> root.getElementsByTagName( ' programming ')
[<DOM Element : programming at 137788492 > ,
 <DOM Element : programming at 137755692 > ,
 <DOM Element : programming at 137602140 >]
```

4.2.5. `NodeList` és szekvencia objektumok

A **`childNodes`** és **`getElementsByTagName()`** és **`NodeList`** függvények visszatérési értékei « klasszikus »szekvenciaként viselkednek.

Ahhoz, hogy egy objektum szekvenciaként viselkedjen (minimum) a következő metódusokkal kell rendelkeznie :

- **`len()`**__ és __**`getitem(i)`**__ az elemekhez való hozzáféréshez (**`for`** ciklus),
- _____**`setitem(i)`**__ és __**`delitem(i)`**__ a módosítható szekvenciák számára.

4.3. Hozzáférés egy csomópont információihoz

4.3.1. Csomóponthoz kötött információk

`nodeType` a csomópot reprezentáló konstanst (1-l0) adja meg.

```
ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE,
ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE,
DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE.
 >>> root . nodeType == minidom . Node . ELEMENT_NODE
 1
 >>> root.firstChild.nodeType == minidom.Node.ELEMENT_NODE
 0
 >>> root.firstChild.nodeType == minidom.Node.TEXT_NODE
 1
```

`nodeName` a csomópont (tag) nevét adja meg. Figyelem, a viselkedése a csomópontoktól függően változik : egy elem esetén értéke az XML tag neve lesz, szöveget reprezentáló csomópont esetén **`#text`**.

```
>>> root.nodeName
u'contacts '
>>> root.firstChild.nodeName
'# text '
```

`nodeValue` a csomópont értékét (tartalmát) adja meg. Figyelem, a viselkedése a csomópontoktól függően változik : egy elem esetén értéke **`None`**, az információt tartalmazó csomópont esetén az információ.

```
>>> root.nodeValue
>>> root.firstChild.nodeValue
u'\n '
>>> root.firstChild.data
u'\n '
```

4.3.2. Egy csomópont attribútumaihoz kötött információk

`hasAttributes()` ellenőrzi, hogy vannak-e attributumai egy csomópontnak.

```
>>> root.hasAttributes()
0
>>> root.childNodes[1].hasAttributes()
1
```

attributes a csomópont attrbutumai (NamedNodeMap vagy None típusú objektum.

```
>>> root.firstChild.attributes
>>> root.childNodes[1].attributes
<xml.dom.minidom.NamedNodeMap object at 0 x81b298c >
```

getAttribute () visszatérési értékül egy attributum értékét **Attr** típusú objektumként adja meg.

```
>>> c = root.childNodes[1]
>>> print c.getAttribute( 'name ')
doe
```

getAttributeNode () visszatérési értékül egy attributum értékét karakterláncként adja meg.

```
>>> print c.getAttributeNode( 'name ')
<xml.dom.minidom.Attr instance at 0 x8360424 >
```

Egy **NamedNodeMap** úgy viselkedik, mint egy szótár. Az **item()** metódussal az attributumnév/attributum szövegérték párokhoz, a **keys()** metódussal az attributumok nevéhez, a **values()** metódussal az attributumokat reprezentáló **Attr** objektumok listájához férhetünk hozzá.

```
>>> attr.items()
[(u'name ' , u'doe ') , (u'firstname ' , u'john ')]
>>> attr.keys()
[u'name ' , u'firstname ']
>>> attr.values()
[<xml.dom.minidom.Attr instance at 0 x302ffd28 >,
 <xml.dom.minidom.Attr instance at 0 x302ffd78 >]
>>> for a in attr.values():
        print a. nodeType , a. nodeName , a. nodeValue
...
2 name doe
2 firstname john
```

A szótárak klasszikus metódusain kívül két metódus szolgál a NamedNodeMap -ek manipulálására :

- **length** megadja az attributum-map hosszát
- **item(i)** visszatérési értéke a map i-edik (**Attr** típusú) eleme adja vagy **None**

```
>>> attrs = root.childNodes[1].attributes
>>> for idx in range (0 , attrs . length ):
        a = attrs.item( idx)
        print '(' + a. name + ')',
        print a.nodeType , a.nodeName , a.nodeValue
...
( name ) 2 name doe
( firstname ) 2 firstname john
```

4.4. Egy XML dokumentum konstruálása

4.4.1. Egy DOM-fa létrehozása

Egy dokumentum (a DOM-fájának a létrehozása) a Document osztály (ez egy olyan objektum mint a többi) példányosításával történik. Ez az osztály adja a csomópontok előállítására szolgáló metódusokat. Egy dokumentum a létrehozásakor üres, nem tartalmaz még gyökércsomópontot sem.

```
>>> newdoc = minidom.Document()
>>> newdoc
<xml.dom.minidom.Document instance at 0 x827fe7c >
>>> newdoc.documentElement
>>>
```

4.4.2. DOM-csomópontok létrehozása

A csomópontokat létrehozó metódusok nem tartalmazzák a fában létrehozott elem hozzáadását a csomóponthoz. A következőkben létre hozunk egy rendkívül egyszerű dokumentumot :

- egy **`root`** gyökeret, aminek **`name`** a neve
- egy kommentet
- egy sometext csomópontot, ami egy szövegcsomópontot tartalmaz

`createElement(tag)` egy új (**`Element`** típusú) elemet hoz létre a paraméterként megadott tag-névvel és ezt adja meg visszatérési értékként.

```
>>> newroot = newdoc.createElement( 'root ')
```

`createAttribute(name)` egy **`name`** nevű **`Attr`** típusú csomópontot hoz létre. Ha már létrehoztuk a csomópontot, az értéket a **`nodeValue`** értékkel lehet beállítani.

```
>>> rootattr = newdoc.createAttribute( 'name ')
>>> rootattr.nodeValue = 'foo '
```

`createTextNode(data)` egy adatcsomópontot hoz létre, ami a paraméterként átadott szöveget tartalmazza (á inclure dans un noeud englobant, ici le noeud sometext créé pour)

```
>>> textnode = newdoc.createElement( 'sometext ')
>>> text = newdoc.createTextNode( 'this node \ncontains text ')
```

`createComment(text)` egy kommentcsomópontot hoz létre, ami a paraméterként átadott szöveget tartalmazza.

```
>>> comment = newdoc.createComment( 'a very usefull comment ')
```

4.4.3. Csomópontok hozzáadása egy fához

A csomópont hozzáadó metódusok a csomópontkreáló metódusok kiegészítői. *Leur dissociation est d˙ue aux multiples usages que l'on peut faire d'un noeud*. Egy dokumentumot létrehozhatunk egy létező XML dokumentumból. Ebben az esetben nem fogunk újra létrehozni minden csomópontot, hanem esetleg újraszervezzük őket.

appendChild(new) egy csomópont gyermekcsomópontjainak listája ad fűz egy elemet.

```
>>> newdoc.appendChild( newroot )
<DOM Element : root at 137797476 >
>>> textnode.appendChild( text )
<DOM Text node " this node
c... ">
```

insertBefore(new, old) egy csomópont adott gyermekcsomópontja elé fűz be egy elemet.

```
>>> newroot.insertBefore( comment , textnode )
<DOM Comment node "a very use ...">
```

replaceChild(new, old) egy csomópont gyermekcsomópont elemét helyettesíti egy másik elemmel.

setAttribute(name, value) egy csomópont számára létrehoz egy új attributumot anélkül, hogy egy **Attr** típusú példánnyal adnánk át

```
>>> newroot.setAttribute( 'usefull ' , 'nop ')
```

setAttributeNode(new) az adott csomóponthoz egy attribútumot kapcsol.

```
>>> newroot.setAttributeNode( rootattr )
```

4.4.4. Egy DOM-fa attribútumainak törlése

Lehetőség van az attribútumok törlésére is egy DOM-fából például azért, hogy egy új, kifinomultabb verziót hozzunk létre, vagy egy XML dokumentum újrastruktúrálása érdekében.

removeChild(old) törli egy csomópont gyermekcsomópontjait. Az **unlink()** kel együtt kell alkalmazni.

```
>>> try:
        old = root.removeChild( root.firstChild )
        old.unlink()
    except ValueError : print 'failed '
    ...
```

removeAttribute(name) a neve alapján törli egy csomópont egy attribútumát.

```
>>> root.firstChild
<DOM Element : contact at 137757692 >
>>> root.firstChild.removeAttribute( 'firstname ')
```

removeAttributeNode(old) törli egy csomópont egy attribútumát az attribútumot reprezentáló **Attr** hivatkozás alapján vagy **NotFoundErr** kivételt generál.

4.4.5. Egy XML dokumentum serializálása

Egy XML dokumentum létrehozásával az a célunk, hogy egy fájl-t kapjunk. A **`toxml()`**, **`toprettyxml()`** metódusok egy DOM-fának az (egysoros vagy többsoros behúzásos) szövegváltozatát állítják elő. Mindegyik csomópontra hívhatók és a kiértékelés automatikusan rekurzív. Egy dokumentum serializálásához elegendő ezen metódusokegyikét hívni a dokumentum objektumra. Ha a metódushívás egy csomóponton történik, akkor a fájl a dokumentum egy részhalmaza lesz.

```
>>> newdoc . toxml ()
' <? xml version ="1.0" ? >\n< root name =" foo " usefull =" nop " > \
<!--a very usefull comment --> \
<text > this node \ ncontains text </ text ></ root >'
>>> output = open ( '/tmp /tmp.xml ' , 'w')
>>> output . write ( newdoc . toprettyxml ())
>>> output . close ()
```

4.5. Gyakorlatok

Az etudiants.xml fájl 400 hallgató (véletlenszerűen generált) adatait tartalmazza az alább leírt struktúrával. A gyakorlat célja 3 darab XML formátumú lista file létrehozása fájl-okban, melyek rendre a megfelelt, kiegészítő szakos és elutasított hallgatók adatait tartalmazzák.

```
<?xml version ="1.0" ?>
<etudiants >
   <etudiant nom="doe" prenom =" john " dossier ="0">
      <origine universite ="mit" />
      <formation discipline =" informatique " niveau ="4" />
      <resultats moyenne ="16" />
   </ etudiant >
</ etudiants >
```

A három listához tárolt információk : a családnév, a keresztnév, az egyetem és az eredmény. A kiválasztási szabályok a következők :

- Ahhoz, hogy egy hallgató megfeleljen Bac + 4 (érettségi + 4 év ráképzés) szinten kell lennie informatikai képzésben, az átlagának 12 fölött kell lenni.
- Ha a tudományág nem informatika, akkor a kiegészítő listán legyen rajta (az átlagra vonatkozó feltétel továbbra is fennáll).
- Egyébként el van utasítva.

A feladat megoldásához elemezni kell az etudiants.xml fájl-t, navigálni kell az így létrehozott DOM-fában és három DOM-fát kell készíteni, amiket az adatfeldolgozás végén az **`admissibles.xml, complementaires.xml et refuses.xml`** fájl-okba serializálunk. Az említett három fájl szerkezete a következő :

```
<?xml version ="1.0" ?>
<admissibles | complementaires | refuses >
  <etudiant nom="doe" prenom =" john " dossier ="0">
    <origine universite ="mit" />
    <resultats moyenne ="16" />
  </ etudiant >
</ admissibles | complementaires | refuses >
```

5. A Python és az adatok tartós tárolása

5.1. DBM fájl-ok

5.1.1. Leírás és alkalmazás

Az **anydbm** modul az egyszerű fájl-ok alkalmazásával megoldást kínál az állandó adattárolás egyszerű igényeire. Egy implementáció független, standard megoldást sugall. Egy DBM fájl-t úgy használunk, mint egy szótárat, tehát a használata viszonylag egyszerű, Csak az inicializálása eltérő (filemegnyitás). Az adatokat karakterláncként manipuláljuk, kulcsok segítségével férhetünk hozzájuk.

open(name, c) megnyitja a name nevű fájl-t (a 'c' a fájl létrehozását jelenti, amennyiben a fájl még nem létezik).

```
>>> import anydbm
>>> file = anydbm.open( 'blabla ' , 'c')
```

close() lezárja a fájl-t (az implementációtól függ, hogy megköveteljük-e a használatát. Ha biztosak akarunk benne lenni, hogy elmentettük az adatokat, akkor inkább használjuk explicit módon a **close** -t.).

```
>>> file.close()
```

Az írás és az olvasás úgy történik, mint egy szótár esetén, azonban a kulcsok kötelezően stringek. A következő példa két bejegyzést (**foo** és **bar**) hoz létre egy DBM fájl-ban, a fájl-t visszaolvassa és lezárja.

```
>>> file = anydbm.open( 'blabla ' , 'c')
>>> file[ 'foo '] = 'perrier c foo '
>>> file[ 'bar '] = 'cd bar ; more beer '
>>> print file[ 'foo ']
perrier c foo
>>> print file[ 'bar ']
cd bar ; more beer
>>> file.close()
```

has_key() teszteli egy kulcs létezését.

```
>>> file = anydbm.open( 'blabla ' , 'c')
>>> file.has_key( 'foo ')
1
```

keys() a fájl kulcsainak egy listáját adja visszatérési értékül. A kulcsok előfeltételét képezik egy DBM fájl bejárásának.

```
>>> file.keys()
['foo ' , 'bar ']
>>> for key in file . keys(): print key , file[ key]
foo 'perrier c foo '
bar 'cd bar ; more beer '
```

len() magadja a fájl-ba tett bejegyzések számát.

```
>>> len( file )
2
```

del lehetővé teszi egy bejegyzés törlését (ez megfelel egy standard szótár működésének).

```
>>> file.keys()
['foo ' , 'bar ']
>>> del file[ 'foo ']
>>> file.keys()
['bar ']
>>> len( file )
1
>>> file.close()
```

5.1.2. Korlázozások

A DBM fájl-ok kizárólag karakterláncok tartós tárolását teszik lehetővé. A karakterlánccá illetve a visszakonvertálást manuálisan kell elvégezni, ami kompozit objektumokra gyorsan összetetté teszi az alkalmazását.

5.2. Pickle és Shelve

5.2.1. Object pickling

A **pickle** modul (ez egy standard modul) lehetővé teszi a memória objektumok serializálását (és deserializálását). Hasznos az adatok tartós tárolásákor és a hálózaton át történő adatátvitelkor.

```
>>> import pickle
```

Az adatmanipuláció serializálási aktusonként egy fájl-t használ, amiben az adatokat stringek formájában vannak tárolva. Az adatok nincsennek struktúrálva (következésként nincs gyors keresés).

5.2.2. (De)serializálás és fájl-ok

A Pickler osztály fájl-ba serializál és a dump() metódus valósítja meg egy objektum serializálását (a modulban létezik egy ekvivalens függvény is). A következő példa a **foo.saved** fájl-ba serializál egy szótárat.

```
>>> foo = { 'a': 'aaa ' , 'b': 'bbb ' , 'c': 'ccc '}
>>> output = open( 'foo.saved ' , 'w')
>>> p = pickle.Pickler( output )                    # (1)
>>> p.dump( foo )                                   # (2)
>>> output.close()
```

vagy pedig

```
>>> output = open( 'foo . saved ' , 'w')
>>> pickle.dump( foo , output )                     # (1 ,2)
>>> output.close()
```

Az Unpickler osztály egy fájl-ból deserializál, a load() metódus deserializál egy objektumot (a modulban létezik egy ekvivalens függvény is). A következő példa az előző példában serializált szótárat visszatölti a **foo.saved** fájl-ból.

```
>>> input = open( 'foo . saved ' , 'r')
>>> p = pickle.Unpickler( input )                   # (1)
>>> foo2 = p.load()                                 # (2)
>>> input.close()
>>> foo2
{'a': 'aaa ' , 'b': 'bbb ' , 'c': 'ccc '}
```

vagy pedig

```
>>> input = open ( 'foo . saved ' , 'r')
>>> foo2 = pickle . load ( input )                    # (1 ,2)
>>> input . close ()
```

5.2.3. (De)serializálás és karakterláncok

dumps() egy karakterláncba serializál egy objektumot (nem pedig egy fájl-ba). Ez egy nagyon praktikus művelet például a hálózaton keresztül történő üzenetváltáshoz.

```
>>> data = pickle.dumps( foo)
>>> data
"(dp0 \nS 'a '\np1 \nS 'aaa '\np2 \nsS 'c '\np3 \nS 'ccc '\np4 \
          \nsS 'b '\ np5\nS 'bbb '\ np6\ns."
```

loads() deserializál egy karakterláncot egy objektummá.

```
>>> foo3 = pickle.loads( data )
>>> foo3
{'a': 'aaa ' , 'c': 'ccc ' , 'b': 'bbb '}
```

5.2.4. DBM + Pickle = Shelves

A shelves modul a két előző modult használja ki (a második interface-ét felkínálva) :

- **pickle** (esetlegesen komplex) objektumok serializálásához
- **anydbm** a kulcsok és a fájl-ok kezeléséhez

A következő példa egy **base** fájl-t hoz létre, amiben a **foo** kulcshoz asszociálva egy szótárat tárolunk. Aztán újra megnyitjuk ezt a fájl-t, hogy kinyerjük ebből a könyvtárból az **a** kulcshoz asszociált bejegyzést.

```
>>> import shelve
>>> base = shelve.open( 'base ')
>>> base[ 'foo '] = { 'a': [ 'a1 ','a2 '], 'b': [ 'b1 ','b2 ']}
>>> base.close()
>>> base2 = shelve.open( 'base ')
>>> print base2[ 'foo '] [ 'a']
['a1 ','a2 ']
>>> base2.close()
```

5.2.5. Megjegyzések

- A konkurens update-elést nem támogatja a **shelve**, az **fcntl** alaklmazása az egyik lehetőség.
- Az osztálydefiníciónak importálhatóknak kell lenni, amikor a **pickle** betölti az adatokat.
- Két **DBM** implementáció között nincs garantálva a kompatibilitás.

5.3. A Python és az SQL

5.3.1. Példák Postgres-sel

A Pythonban van egy modul, ami (de facto) standardizált interface-t nyújt az adatbázisokhoz. A különböző kurrens adatbázisokhoz léteznek implementációi. Minden esetre minden adatbázis alkalmazás eltérhet,tehát az adatbázishoz való hozzáférés Python kódja *nem 100%-osan portábilis*[19].

Most egy Postgres adatbázissal illusztrálok egy adatbázis alkalmazást[20]. A psycopg modul biztosít hozzáférést a Pythonból egy Postgres adatbázishoz. Ez lehetővé teszi, hogy jó kompromisszumot találjunk a nyelv és az adatbázis kezelő teljesítménye között : az adatkezelés egy részét SQL-ben valósítjuk meg, míg egy másik részét Pythonban.

Adatbázis létrehozása

- az adatbázis létrehozása a filerendszerben (egy alkalommal)
- a server elindítása (minden egyes alkalommal, amikor elindítjuk a gépet)
- az adatbázis (és ha szükséges a felhasználók létrehozása)
- adminisztrátor = *user* Unix-értelemben, jelszó nélkül

```
$ initdb -D < repertoire >
$ pg_ctl -D < repertoire > start [-o "-i" # a hálózat számára ]
$ createdb test
```

5.3.2. Alapműveletek

`connect()` művelet visszatérési értékül egy adatbázis kapcsolatot reprezentáló objektumot ad. Az argumentumok az adatbázisnak illetve a környezetnek megfelelő változók (például a **`host`** nem kötelező, ha az adatbázis lokális).

```
>>> import psycopg
>>> connexion = psycopg.connect(
    " host = localhost dbname = test user = rafi ")
```

Cursor-okat használunk az adatbázis interakciókhoz : SQL kérések küldése. Ezeket a kapcsolat-objektum hozza létre. A cursor-objektum **`execute()`** metódusa teszi lehetővé egy SQL kérés kiértékelésének kérését az adatbázistól. Most a cursort az – egyetlen – **`val`** egész típusú mezőt tartalmazó **`test1`** adattábla létrehozására használjuk. Utána a kurzort arra használjuk, hogy tíz adatot szúrunk be ebbe a táblába.

```
>>> curseur = connexion.cursor()
>>> curseur.execute(
        "CREATE TABLE test1( val int4 )")
>>> for i in range (10):
        curseur.execute(
            "INSERT INTO test1 VALUES(%d)" , (i ,))
```

Példák beszúrásokra. Egy beszúrást megvalósíthatunk úgy, hogy az értékeket egy tuple-ben (mint az előző példában) vagy egy szótárban adjuk meg. A tuple-k használata esetén az adatokat a tuple-

19 Tehát tanácsos egy alkalmazásban definiálni egy adatbázis absztrakciós réteget.

20 Ugyanez egy kicsit eltérő MySQL-lel, például a felkapcsolódás és a commit- ek kezelése.

beli sorrendben vesszük. A tuple-nek ugyanolyan hosszúnak kell lenni, mint a kérés formátum-tuple-e (a **VALUES** klauzula argumentuma). Akövetkező példa egy új táblát hoz létre és egy szótárat alkalmaz a beszúrandó értékek megadására. Az **executemany** metódussal Python ciklus írása nélkül szúrhatunk be egy adatsorozat a táblába.

```
>>> curseur = connexion.cursor()
>>> curseur.execute( '''CREATE TABLE test2( name text ,
        firstname text )''')
>>> curseur.execute( '''INSERT INTO test2
        VALUES(%( name )s , %( firstname )s)''',
        {'name ': 'doe ' , 'firstname ': 'john '})
>>> valeurs = (( 'martin ' , 'pierre ') , ( 'dupont ' , 'paul '))
>>> curseur.executemany('''INSERT INTO test2
        VALUES(%s , %s)''' , valeurs )
```

A kérések eredményei (a **SELECT** parancsra adott válaszok) Python adatszerkezetek : tuple-k (ezek egy válasz adatai) listái (ez utóbbiak válaszok együttesei). A cursor **fetchone()** és **fetchall()** műveletei lehetővé teszik, hogy egy válaszhoz (valójában a válaszokhoz egyessével), vagy az összes válaszhoz egyszerre férjünk hozzá[21]. A következő példa végrehajt egy kérést az adatbázison, majd először kinyeri az első választ, kiírja az első mezőt, utána kinyeri az összes maradék választ és kiírja a két mezőt.

```
>>> curseur.execute(" SELECT * FROM test2 ")
>>> prem = curseur.fetchone()
>>> print prem[0]
doe
>>> valeurs = curseur.fetchall()
>>> for v in valeurs :
        print v[0] , v[1]
...
martin pierre
dupont paul
```

close() explicit módon kéri egy kapcsolat lezárását (ez implicit, ha a kapcsolat-objektum Python értelemben van lerombolva).

commit() jóváhagy egy adatbázis tranzakciót (a kapcsolat-objektum metódusa).

rollback() töröl egy adatbázis tranzakciót (a kapcsolat-objektum metódusa).

autocommit(1|0) a műveleteket az autocommit aktiválja vagy deaktiválja (a kapcsolat-objektum metódusa).

5.3.3. Dátumok kezelése

A (**psycopg** -hez kapcsolódó) **mx** modul olyan osztályokat tartalmaz, elsődlegesen a DateTime osztályt, amik a dátumok és ***timestamp-ek*** kezelésére szolgálnak.

```
>>> import mx
>>> mx.DateTime.now()
<DateTime object for '2003 -06 -12 18:47:36.54 ' at 8 aa0a30 >
>>> t1 = psycopg.TimestampFromMx(mx. DateTime.now())
>>> t2 = psycopg.Timestamp(2003 , 06 , 15 , 15 , 32 , 07)
```

21 Ügyeljünk a sok választ adó **SELECT** -ekre, a **fetchall()** -nak van egy felső határa.

5.3.4. Bináris adatok kezelése

A bináris adatok (például képek vagy hangok) használatát a beszúrásukkor a **`Binary`** osztállyal explicit módon kell deklarálni. A következő példa egy adattáblát hoz létre, amiben fotókat és a fotók nevét fogjuk tárolni, majd beszúr egy fotót.

```
>>> curseur = connexion.cursor()
>>> curseur.execute('''CREATE TABLE test3( id int4 ,
        name text , photo bytea)''')
...
>>> curseur.execute('''INSERT INTO test3
    VALUES(%d , %s , %s)''',
    (1 , 'shoes.jpg ' , psycopg.Binary( open('shoes .jpg ').read())))
```

A bináris adatok kinyerése hasonlít a klasszikus adatok kinyeréséhez. A következő példa az adatbázisban tárolt fotót nyeri ki és elmenti egy **`new_shoes.jpg`** fájl-ba.

```
>>> curseur = connexion.cursor()
>>> curseur.execute( 'SELECT * FROM test3 ')
>>> val = curseur.fetchone ()
>>> open( 'new_ ' + val [1] , 'wb ').write( val [2])
```

5.4. Gyakorlatok

5.4.1. Az MVC « Model » modulja

Írjunk egy Python scriptet, ami létrehoz egy hallgatókat reprezentáló adattáblát, melynek a következők a mezői : ügyiratszám (int4), családnév (text), utónév (text), egyetem (text), szak (text), szint (int4), átlag (int4),.

Írjunk egy Python programot, ami betölti az **`etudiants.xml`** fájl adatait ebbe az adattáblába. Ehhez felhasználható az előző gyakorlat kódjának egy része, ami a DOM-fa bejárását végzi.

5.4.2. Az MVC « Controller » modulja

Írjunk három osztályt, melyeknek az eval() metódusa a következő műveleteket végzi az adatbázison :

- megadja a hallgatók ügyiratszámát, családnevét és utónevét.
- megadja az ügyiratszámhoz a hallgató teljes adatlapját.
- beszúr egy hallgatói adatlapot, az összes adat megadásával.

Írjunk egy Controller osztályt, ami a három előző osztály kompozíciója és aminek metódusai vannak a példányokhoz való hozzáféréshez a végrehajtás során (tegyenek lehetővé olyan interakciókat, mint amilyeneket az előző példában írtunk le).

```
ctrl = Controleur(...)
list_all_ctrl = ctrl.get_list_all_ctrl()
student_list = list_all_ctrl.eval()
```

6. A Python és a grafikus interface-ek

6.1. A Python és a Tkinter

6.1.1. Tkinter

A Tkinter modul [3] a J. Ousterout által Tcl-ben [10] – grafikus interface-ek készítésére – fejlesztett Tk library-n alapul és számos platformon (egyebek között X11, MS-Windows, Macintosh) rendelkezésre áll. A Tkinter a Python de facto standardja, ami a Tk objektum-vízióját nyújtja. Minden esetre a Tcl egy olyan nyelv, amiben minden string. Ebből eredően a Tkinter-ben használt számos érték karakterlánc (amiket közvetlenül így, vagy a modul által kínált változókba csomagolva használunk).

Mint minden grafikus interface-ű program esetében, a programvégrehajtást az események vezérlik. A Python és a grafikus interface közötti interakciók több formát vehetnek fel :

- Python GUI művelet → Tkinter → Tk → grafikus könyvtár
- Grafikus esemény → Tk → Tkinter → Python műveletek

6.1.2. Az első lépések

A « Hello world » grafikus verziója négy sor, amit a következő példa mutat be. Betöltjük a Tkinter modult. Létrehozunk egy widget-et (jelen esetben egy **`Label`**-t), majd hozzákapcsoljuk a grafikus környezethez (**`pack`**). Végül elindítjuk az eseményfigyelőt. Ez mindaddig aktív, amíg be nem zárjuk az ablakot (és az interpreter promptja fel van függesztve)

```
>>> import Tkinter
>>> widget = Tkinter.Label( None , text ='hello world !')
>>> widget.pack()
>>> widget.mainloop()
```

6.1.3. Widget-ek konfigurálása

A « Hello world » **`Label`** widget-e esetében nincs szülőablak (az első argumentum **`None`**). Közvetlenül az alapértelmezett ablakhoz van kapcsolva. A **`Label`** text reprezentálja a konfigurációját. A widget konfigurálható vagy a létrehozása után a konfigurálása megváltoztatható a **`config()`** metódussal. A következő példa az eredmény vonatkozásában megegyezik az előző példával.

```
>>> widget = Tkinter.Label( None )
>>> widget.config( text ='hello world !')
>>> widget.pack()
>>> widget.mainloop()
```

6.1.4. Widget-ek konfigurálása a pack() metódussal

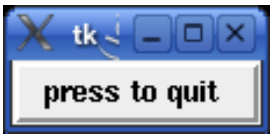
A widget-ek komponálása a geometria manager-hez van delegálva.

- A widget elhelyezése a konténeréhez képpest (a **`side`** konfigurációs opcióval) a fő irányok (Tkinter.TOP, Tkinter.LEFT, Tkinter.RIGHT, Tkinter.BOTTOM) alkalmazásával történik. Egy widget alapértelmezetten a létező widget-ek tetejéhez, vagy aljához van kapcsolva.
- Ha maximálisan ki akarjuk tölteni a teret, alkalmazzuk az **`expand = YES`** konfigurációs opciót. A **`fill`** opció határozza meg a maximális kitöltés irányát. Ez utóbbi szélességben (Tkinter.X(, magasságban (Tkinter.Y) és mindkét irányban (Tkinter.BOTH).

6.1.5. Parancsok kiadása

- Ahhoz, hogy egy grafikus interface-ről egy akciót kezdeményezzünk, az aktuális metódusnak definiálni kell egy gombot, egy menüt, egy görgetősort, stb. és hozzá kell kapcsolni egy eseménykezelőt (egy argumentum nélküli függvényt vagy metódust). A következő példa létrehoz egy gombot és hozzákapcsolja a **`sys.exit`** parancsot, ami a programból (esetünkben az interpreterből) történő kilépésre való. A text opció határozza meg a gombon megjelenő szöveget.

```
>>> import sys
>>> import Tkinter
>>> widget = Tkinter.Button( None )
>>> widget.config( text =’press to quit ’ , command =sys.exit )
>>> widget.pack()
>>> widget.mainloop()
```



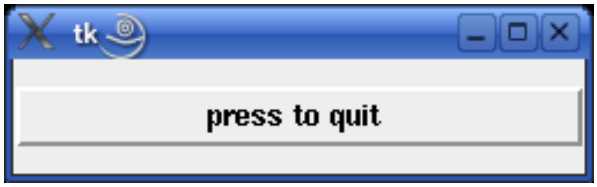
6.1.6. Komponálás és átméretezés

A komponálás és az átméretezés közötti kapcsolat :

- alapértelmezetten a widget megtartja kiindulási méretét,
- a változások követéséhez használjuk az **expand** opciót.

A következő példa úgy konfigurálja a gombot, hogy az ablak átméretezésekor szélességében mindig kitölti az egész teret. Viszont magasságban nem tölti ki az egész teret.

```
>>> import Tkinter
>>> import sys
>>> widget = Tkinter.Button( None )
>>> widget.config( text ='press to quit ' , command =sys.exit )
>>> widget.pack( expand = Tkinter.YES , fill = Tkinter.X)
>>> widget.mainloop()
```



6.1.7. Gombok és műveletek

Egy gomb csak egy akciót indíthat el, ezért gyakran van szükség egy olyan függvény alkalmazására, ami csoportosítja az akciókat. A következő példában a **`handler`** függvény kiíratja a « Hello world » szöveget a standard kimenetre mielőtt kilép az alkalmazásból.

```
>>> import Tkinter
>>> import sys
>>> def handler ():
        print 'hello world !'
        sys.exit ()

>>> widget = Tkinter.Button( None )
>>> widget.config( text ='press to quit ' , command = handler )
>>> widget.pack( expand = Tkinter.YES , fill = Tkinter.X)
>>> widget.mainloop()
```

Egy grafikus widget felhasználói osztálya a megfelelő eszköz az adatkezelés és a widget közötti szoros kapcsolat kialakítására. Ez az együttes (gomb, adatok, akciók) egyszerűen újrafelhasználhatók. Egy alkalmazásból való kilépésre szolgáló gombra mutatok példát, mely esetében az üzenet paraméterezhető.

```
>>> import Tkinter
>>> import sys
>>> class HelloQuitButton :
        def __init__( self , msg='Quit'):
            self.msg = msg
            b = Tkinter.Button( None )
            b.config( text = self.msg , command = self.handle )
            b.pack()
        def handle( self ):
            print self.msg
            sys.exit()

>>> hqb = HelloQuitButton ()
>>> Tkinter.mainloop ()
```

Egy ***handler*** definiálható objektumként : egy függvényt reprezentáló osztály definíciója és példányosítása az alkalmazáshoz. A következő példa az előző handlert osztályként definiálja újra.

```
>>> import Tkinter
>>> import sys
>>> class Callable :
        def __init__( self ):
            self .msg = 'hello world !'
        def __call__( self ):
            print self .msg
            import sys
            sys. exit()

>>> widget = Tkinter.Button( None )
>>> widget.config( text ='hello ' , command = Callable())
>>> widget.pack()
>>> Tkinter.mainloop()
```

6.1.8. Binding definíciók

A **bind()** metódussal lehet egy függvényt egy eseményhez és egy widget-hez kapcsolni. A leggyakoribb események az egérkattintások és a billentyűzetről történő adatbevitel. A következő példa két függvényt hoz létre. Az egyik kiíratja a « Hello world » -öt, a másik a programból történő kilépéskor « bye » -t írat ki a standard kimenetre. Majd létrehoz egy címkét (label) és hozzákapcsolja a **hello** függvényt a balegérgombbal való egyszeri kattintás esetére és a **quit** függvényt a dulpakattintás esetére.

```
>>> import Tkinter
>>> import sys
>>> def hello(event):
        print 'hello world !'

>>> def quit(event):
        print 'bye '
        sys. exit ()

>>> widget = Tkinter.Label( None , text ='press ')
>>> widget.pack()
>>> widget.bind( '<Button -1> ' , hello )
'805810704 hello '
>>> widget.bind( '<Double -1> ' , quit )
'805810224 quit '
>>> widget.mainloop ()
```

A handler által « fogadott »esemény tartalmazza azokat az információkat, amik a művelet keretében interpretálhatók. A következő példa két függvényt hoz létre. Az egyik kiíratja a billentyűzeten beírt karaktert, a másik kiíratja az egérkurzor koordinátáit. A **Tk** osztály reprezentálja a program főablakát. A létrehozott címke ehhez az ablakhoz van kapcsolva (a constructor első paramétere). Majd a két függvény a billentyűzethez és a címkén történő egérkattintáshoz van kapcsolva. A **focus** alkalmazása a címkét választja ki a billentyűesemények fogadására. Végül a főablakon elindítjuk az eseményfigyelő programhurkot. Amikor aktív az ablak és megnyomunk egy billentyűt, akkor ez egy eseményt generál, ami átadódik az **onKey** függvénynek, ami meghatározza az eseményhez kapcsolódó karaktert. (Ugyanez az el alkalmazható az egérkattintásra.)

```
>>> import Tkinter
>>> def onKey( event ):
        print 'got key ', event.char

>>> def onClick( event ):
        print event.widget, event.x ,event.y

>>> root = Tkinter.Tk()
>>> lab = Tkinter.Label( root , text ='hello world ')
>>> lab.bind( '<KeyPress >' , onKey )
'805787968 onKey '
>>> lab.bind( '<Button -1> ' , onClick )
'808388888 onClick '
>>> lab.focus()
>>> lab.pack()
>>> root.mainloop()
.805789368 30 14
.805789368 44 11
got key e
got key r
```

Néhány gyakori binding név

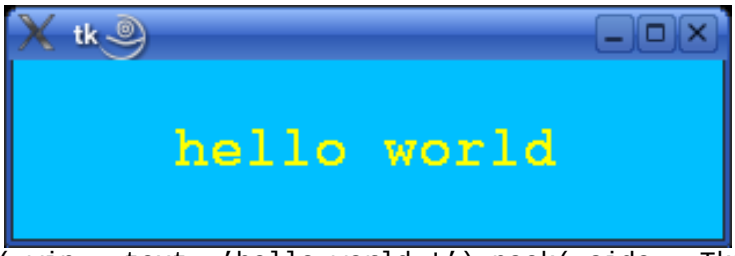
<KeyPress>	Egy billentyű lenyomása
<KeyPress-a>	A kis 'A' lenyomása (kisbetű)
<KeyPress-A>	A nagy 'A' lenyomása (nagybetű)
<Return>	Enter lenyomása
<Escape>	Escape lenyomása
<Up> <Down>	Nyílbillentyűk lenyomása
<Button-1>	Kattintás a balegérgombbal
<Button-2>	Kattintás a középső (vagy mindkét) egérgombbal
<Button-3>	Kattintás a jobbegérgombbal
<ButtonRelease>	A balegérgomb kattintás vége
<Motion>	Az egér mozgatása
<B1-Motion>	Az egér mozgatása balegérgomb kattintással
<Enter> <Leave>	Az egér egy widget fölé mozog, vagy fölül mozog
<Configure>	Ablak átméretezése
<Map> <Unmap>	Ablak megnyitása és ikonná tétele

6.1.9. Widgetek csoportosítása

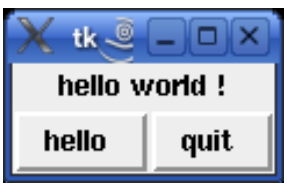
A widget-ek csoportosítása grafikus *konténerek* segítségével történik, leginkább **Frame**-ek definiálásával, amikhez hozzákapcsoljuk a widget-eket. A következő példa egy **Frame**-et használ két gomb csoportosításához. A második gombhoz kapcsolt parancs teszi lehetővé az ablak bezárását anélkül, hogy leállítanánk az interpretert.

```
>>> import Tkinter
>>> def hello():
        print 'hello world !'

>>> win = Tkinter.Frame()
>>> win.pack ()
```




```
>>> Tkinter.Label( win , text ='hello world !').pack( side = Tkinter.TOP)
>>> b1 = Tkinter.Button( win , text ='hello ' , command = hello )
>>> b1. pack( side = Tkinter.LEFT )
>>> b2 = Tkinter.Button( win , text ='quit ' , command =win.quit )
>>> b2.pack( side = Tkinter.RIGHT )
>>> win.mainloop ()
```



6.1.10. Objektum orientált widget-ek

Az objektum orientált megközelítés jól alkalmazható a grafikus interface-ekre az öröklést specializálás vagy kiterjesztés útján alkalmazva új widget-ek definiálására. A következő példa bemutatja, hogyan specializálunk egy Button widget-et egy olyan gombbá, amivel kiléphetünk az alkalmazásból (vagyis a gomb nem rombolja le az alkalmazást).

```
>>> import Tkinter
>>> class MyByeButton( Tkinter.Button ):
        def __init__( self , parent =None , ** config ):
            Tkinter.Button.__init__( self , parent , config )
            self.pack()
            self.config( command = self.callback )
        def callback( self ):
            print 'bye ... '
            self.destroy()
>>> MyByeButton( text ='hello world ')
>>> Tkinter.mainloop()
```

6.1.11. A megjelenés apropóján

A szín és a méret minden Tkinter widget számára megadható. A fontkészlet, a karakterek mérete és stílusa minden szöveget tartalmazó widget számára megadható. A következő példa egy sárga színű, kék hátterű, 20 pixel méretű, Courier típusú, félkövér (bold) stílusú feliratos, 3 sor magasságú és 20 karakter széles címkét definiál.

```
>>> import Tkinter
>>> root = Tkinter.Tk()
>>> msg = Tkinter.Label( root , text ='hello world ')
>>> msg.config( font =( 'courier ' , 20 , 'bold '))
>>> msg.config( bg='deepskyblue ' , fg='yellow ')
>>> msg.config( height =3 , width =20)
>>> msg.pack( expand = Tkinter.YES , fill = Tkinter.BOTH )
>>> root.mainloop()
```

6.1.11.1 Többablakos alkalmazás

Ugyanabban az alkalmazásban lehetőség van független ablakok (amiket nem tartalmaz vízuálisan ugyanaz az ablak)definiálására. A következő példa két független ablakot definiál, melyek rendre a « hello» és a « world» címkéket tartalmazzák.

```
>>> import Tkinter
>>> root = Tkinter.Tk()
>>> win1 = Tkinter.Toplevel( root )
>>> Tkinter.Label( win1 , text ='hello ').pack()
>>> win2 = Tkinter.Toplevel( root )
>>> Tkinter.Label( win2 , text ='world ').pack()
```

Egy többablakos alkalmazás befejezése :

- **`destroy()`** rekurzívan törli az érintett ablakokat,
- **`quit()`** az ablak lerombolása nélkül befejezi az eseményfigyelő hurkot.

```
>>> root = Tkinter.Tk()
>>> win1 = Tkinter.Toplevel( root )
>>> b1 = Tkinter.Button( win1 )
>>> b1.config( text ='moi ' , command = win1 . destroy )
>>> b1.pack()
>>> win2 = Tkinter.Toplevel( root )
>>> b2 = Tkinter.Button( win2 )
>>> b2. config( text ='nous ' , command = root.destroy )
>>> b2.pack()
>>> root.mainloop()
```

6.2. Kalandozás a gyakran használt widgetek között

A fejezet hátralévő részében feltételezem, hogy a **`Tkinter`** az aktuális névtérbe van importálva. Bár biztonságosabban használhatjuk a modulokat, ha a modulnevet prefixumként használjuk, az igaz, hogy interaktív módban meg van az előnye annak, ha mindent globális érvényességi körbe (scope-ba) importálunk. Azonban mindig ügyelnünk kell a névütközések kockázatára.

```
>>> from Tkinter import *
```

6.2.1. Entry : szövegbeviteli mező

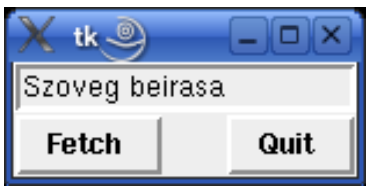
Az Entry widget egy egysoros szövegbeviteli mező. A következő konfigurációs opcióktól függ :

- **`state='disable'`** csak olvasás,
- **`show='*'`** maszkolt beírás.

A következő példában egy interface-t készítünk egy szövegbeviteli mezővel és két gombbal. A return billentyű az adatbeviteli boxban a **`fetch()`** függvényhez van asszociálva. Az első gomb kiíratja a standard kimenetre a szövegbeviteli mező tartalmát. Ez a gomb is a **`fetch()`** függvényt használja. A második gomb bezárja az interface-t.

```
>>> def fetch ():
        print 'Texte : <%s>' % ent.get()

>>> root = Tk()
>>> ent = Entry( root )
>>> ent.insert(0 , 'Szoveg beirasa ')
>>> ent.pack( side =TOP , fill =X)
>>> ent.bind( '<Return >' , ( lambda event : fetch ()))
>>> b1 = Button( root , text ='Fetch ' , command = fetch )
>>> b1.pack( side = LEFT )
>>> b2 = Button( root , text ='Quit ' , command = root.destroy )
>>> b2.pack( side = RIGHT )
>>> root.mainloop()
```



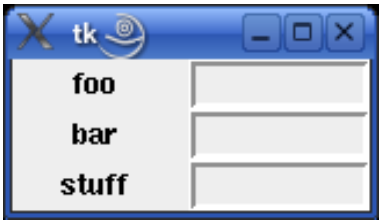
Szöveg manipulációk

- index a szövegbeviteli mezőben : **`0`** -tól **`END`** -ig,
- az **`INSERT`** cursor beszúrási pozíciója,
- **`get(kezdet, vég)`** visszatérési értéke az adott pozícióbeli szöveg,
- **`insert(pos, txt)`** a **`pos`** pozícióba beszúrja a **`txt`** szöveget,
- **`delete(kezdet, [vég])`** a **`kezdet`** és a **`vég`** pozíciók közötti szöveget törli (a **`vég`** paraméter opcionális).

6.2.2. Widget-ek elrendezése

Egy adatbeviteli form típuspélda több **`Label`** és **`Entry`** widget elrendezésére. Ahhoz, hogy egy megfelelően elrendezett formunk legyen, a **`pack`** geometria managert (aminek égtájakat adtunk meg) a « rács-manager »-rel fogjuk helyettesíteni. A **`grid( row=X, column=Y)`** metódussal lehet elrendezni a különböző elemeket a rácsban. A következő példa három szövegbeviteli mezős formot generál.

```
>>> root = Tk()
>>> r = 0
>>> for item in [ 'foo ' , 'bar ' , 'stuff ']:
        l = Label( root , text =item , width =10)
        e = Entry( root , width =10)
        l. grid( row=r , column =0)
        e. grid( row=r , column =1)
        r += 1
>>> root.mainloop()
```



Figyelem : A két geometria managert ne használjuk együtt !

6.2.3. Szövegzóna

A Text widget többsoros string formájú szövegbevitelt és szövegmanipulációt kínál. A következő példa egy szövegbeviteli zónát és egy gombot definiál. Ez utóbbira kattintva a program kiíratja a fetch() függvény segítségével a szövegbeviteli zóna tartalmát a standard outputra. A szövegzóna WRAP=WORD opciója meghatározza, hogy a szövegzóna tartalmának feldarabolása a szóhatárokon történik (amikor egy szó nem tartható egy soron belül, akkor a szövegzóna automatikusan sort vált).

```
>>> def fetch():
        print txt.get( '1.0 ' , END+' -1c')

>>> root = Tk ()
>>> txt = Text( root , height =5 , width =20 , wrap = WORD )
>>> txt.insert( '1.0 ' , 'Szoveg beirasa ')
```

```
>>> txt.pack( side =TOP )
>>> Button( root , text ='fetch ' , command = fetch ).pack()
>>> root.mainloop()
```

Szöveg manipulációk

- index egy szövegzónán : 1.**0** -tól (sor.oszlop) **END** -ig,
- az **INSERT** megadja a cursor beszúrási pozícióját,
- **SEL_FIRST, SEL_LAST** a kiválasztás elejének és végének pozícióját tartalmazza
- **get(kezdet, vég)** visszatérési értéke az adott pozícióbeli szöveg,
- **get('1.0', END+'-1c')** visszatérési értéke az egész szövegzóna tartalma,
- **insert('li.co', txt)** a megadott pozícióba beszúrja a **txt** szöveget,
- **delete(kezdet, [vég])** a **kezdet** és a **vég** pozíciók közötti szöveget törli (a **vég** paraméter opcionális).
- **search(string, kezdet, vég)** visszatérési értéke a string szövegzóna beli kezdő pozíciója
- **see('li.co')**

6.2.4. A Függelék

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful 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. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work uner the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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 Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document 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 fájl format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.
An image format is not Transparent if used for any substantial amount of text. 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, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include 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, PostScript or PDF 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.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty

Disclaimers may have is void and has no effect on the meaning of this License.

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.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.
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.

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:

A. 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.
B. 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 fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. 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. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve 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.
J. 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.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. 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. M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers. 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.

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, and that you preserve all their Warranty Disclaimers.

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".

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.

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, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.
Otherwise they must appear on printed covers that bracket the whole aggregate.

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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

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.

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.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.