Tartsunk lépést a Pythonnal: a 2.2-es változat

A Python 2.2-es változata pótolja a nyelv néhány jól ismert hiányosságát, és pár jól használható új szerkezetet is bemutat.

A Python 2.2-es 2001 végén jelent meg, és a 2.2.1-es hibajavító változatot a PythonLabs és a Zope Co. fejlesztői közösen mostanában készítették el. A 2.2-es tele van új lehetőségekkel és képességekkel, amelyek közül nem egy jelentős előrelépés a nyelvben. Ezek a fejlesztések nagymértékben kitágítják a Python-fejlesztők lehetőségeit.
A Pythonnal egyszerű, ugyanakkor sokrétű fejlesztőeszközhöz jutunk, amely a parancsnyelvek egyszerűségét a gépi kódban futó objektumközpontú nyelvek erejével egyesíti. A Jython segítségével (mely jelölés a Python egy Javára lefordított változatát takarja) a Java-programozók egy új és könnyen használható alkalmazásfejlesztő eszközzel ismerkedhetnek meg.
Ha figyelemmel szeretnéd követni a mindennapi eseményeket, olvasd a PEP-eket, vagyis a Python fejlesztésére irányuló javaslatok gyűjteményét, amelyeket azért hoznak létre, hogy a Python-közösség egyetlen átgondolásra alkalmas ötlete se vesszen kárba. Mielőtt mérlegelni kezdenék egy változtatás szükségességét, tanulmányozzák a változtatással együttjáró gondokat és a lehetséges megoldásokat, valamint azok módszertani részleteit. Nemcsak a PEP-ek pontos részleteit, de jelenlegi állapotukat is pontosan megtudhatod a weboldalon (lásd a ***Kapcsolódó címeket***). Az egyeztetéseket követően a Python-változatok kiadása előtt minden egyes PEP-ről eldöntik, hogy érdemes-e az adott ötletet megvalósítani vagy sem. Például a 2.2-es változat (beleértve a 2.2.x-es alváltozatokat is) elsősorban öt PEP-re épül, melyek a következők: 234, 238, 252, 253 és 255.
A kezdők számára a 2.2-es változattal elkezdődött a Python egész típusainak és kiterjesztett egész típusainak egyesítési folyamata (integer és long integer). Ezentúl az egész számokkal végzett számítások nem okozhatnak túlcsordulást, mivel ha egy eredmény nem fér el az egész típus által megszabott tartományban, önmagától kiterjesztett egész típussá alakítódik.
Az új változattal egyúttal az egymásba ágyazott érvényességi körök is teljesen hatályba léptek, amelyek az eddigi két érvényességi körre épülő rendszert hivatottak leváltani (PEP 227). Bár ez a lehetőség már a 2.1-es változatban megjelent, most már az ott még kötelező `from __future__ import nested_scopes` utasításra sincsen szükség, mivel ez vált az alapértelmezett viselkedésformává. Az Unicode-támogatást UCS-4-re fejlesztették (32-bites előjel nélküli egésztípusok támogatása, bővebben: PEP 261). Kevésbé jelentős változásokon ment keresztül a ***Python Standard Library***, többek között bővült egy új e-mail csomaggal, egy XML-RPC-modullal, a foglalat (socket) modult képessé tették IPv6-kezelésre, és bekerült egy új, okos profiler is.
A 2.2-es változat legjelentősebb újításai a léptetők és az előállítók, amelyek módosítják az osztás műveletjel tulajdonságait, valamint egyesítik a típusokat és az osztályokat.

Léptetők

A léptetők (iterators) segítségével a programozónak lehetősége nyílik sorban végiglépkednie egy adathalmaz elemein. Ez akkor különösen hasznos, ha különböző típusokat tartalmazó lista elemein lépkedünk végig. A Python sorszámozott típusainak (listák, karakterláncok és úgynevezett tuple-ok) a kezelése is meglehetősen egyszerű, akár egy `for` ciklussal végiglépkedhetünk rajtuk anélkül, hogy bármiféle különleges dolgot kellene tennünk. A Python léptetői sorszámozott típusokkal gond nélkül működnek, de az új változattól kezdődően a nem sorszámozott típusok is könnyen kezelhetők, a felhasználó által létrehozott objektumokat is beleértve. A léptetők segítségével bármilyen Python-típuson végiglépkedhetünk.
Nos, ez tényleg jól hangzik, de miért is van szükség léptetőkre a Pythonban? Erre különösen a PEP 234 tér ki, amely ezeket a pontokat említi:

- Bővíthető léptetőfelületet hozni létre.
- A listákon való léptetések frissítéséért.
- A szótártípusok léptetésének jelentős mértékű felgyorsítására.
- Egy valódi léptetőfelület létrehozására ellentétben az eltakaró tagfüggvényekkel, melyeket eredetileg az elemek véletlenszerű elérésére hoztak létre.
- Egy visszirányú együttműködést biztosító felület létrehozásáért, amely a már létező felhasználói osztályokkal és objektumkiterjesztésekkel működik együtt – ezek sorszámozott típusokat és hozzárendeléseket valósítanak meg.

Egy tömörebb és olvashatóbb szerkezet létrehozásáért, amely nem sorszámozott elemeken is képes léptetést megvalósítani (például állományokon és hozzárendeléseken).
Léptetőket kétféleképpen hozhatunk létre: vagy a beépített `iter()` függvényen keresztül, vagy közvetlenül azokon az objektumokon használhatjuk őket, melyekbe ezt a lehetőséget beépítették. Például a listáknak van beépített léptetői felülete, amely nem változott, és a `for` *`mindegyikElem`* `in` *`Lista`* utasításnak megfelelően használható.
Az `iter(objektum)` függvény meghívásával az objektumnak megfelelő léptetőtípushoz jutunk. A léptetőknek egyetlen tagfüggvénye van, a `next()`, ami a következő elemmel tér vissza. Az új `StopIteration` kivétel hivatott jelezni, ha a vezérlés elérte a lista végét.
Természetesen a léptetők használata esetén is számolnunk kell bizonyos korlátozásokkal: nem lehet visszafelé léptetni, nem lehet a készlet elejére ugrani, és a léptetők másolása sem lehetséges. Ha ugyanazon az objektumon szeretnél újra végigugrálni (vagy egyszerre több példányon), létre kell hoznod egy másik léptető objektumot.

Sorszámozott típusok

Mint már említettük, sorszámozott típusok esetén a léptetők az elvárásoknak megfelelően működnek:

```
>>> myTuple = (123, ·xyz·, 45.67)
>>> i = iter(myTuple)
>>> i.next()
123
```

```
>>> i.next()
·xyz·
>>> i.next()
45.67
>>> i.next()
Traceback (most recent call last):
  File "", line 1, in ?
StopIteration
```

Valódi program esetén a fenti kódrészletet egy `try-except` blokkba ágyaztuk volna be. A sorszámozott típusok alapértelmezésben léptetővel rendelkeznek, mint az a `for` ciklus esetén is látható:

```
for i in seq:
    csinalj_valamit(i)
```

Ha jobban megnézzük a fenti utasítást, látható, hogy valójában nem más, mint egy léptetőszerkezet:

```
fetch = iter(seq)
while 1:
    try:
        i = fetch.next()
    except StopIteration:
        break
    csinalj_valamit(i)
```

Mivel a `for` ciklus önműködően meghívja a léptető `next()` tagfüggvényét, ezért a kódon nem szükséges változtatni.
Az `iter()` beépített függvénynek egy másik formája is létezik, az `iter(`*`megh vhat _elem, ırszem`*`)`, amely az előzőekhez hasonló léptetővel tér vissza. A különbség annyi, hogy a léptető `next()` tagfüggvényének a meghívása az egymás után következő elemekért minden alkalommal a *`megh vhat _elem`*`()` függvényt hívja meg, és amennyiben a visszatért érték megegyezik a megadott `ırszem` értékkel, a léptető egy `StopIterator` kivételt dob.

1. lista Egy szótár léptetése

```
>>> legends = { (·Poe·, · r ·):
                        (1809, 1849, 1976),
...             (·Gaudi·, ·Øp tØsz·):
                        (1852, 1906, 1987),
...             (·Freud·, ·pszichoanalitikus·):
                        (1856, 1939, 1990) }
...
>>> for eachLegend in legends:
...     print ·NØv: %s\tFoglalkozÆs: %s· %
            ➥eachLegend
...     print ·   Sz letØsi idı: %s\tHalÆl:
              ➥%s\tAlbum: %s\n· %
              ➥legends[eachLegend]
...
NØv: Freud      FoglalkozÆs: pszichoanalitikus
  Sz letØsi idı: 1856    HalÆl: 1939
                         ➥Album: 1990

NØv: Poe        FoglalkozÆs:  r
  Sz letØsi idı: 1809    HalÆl: 1849
                         ➥Album: 1976

NØv: Gaudi      FoglalkozÆs: Øp tØsz
  Sz letØsi idı: 1852    HalÆl: 1906
                         ➥Album: 1987
```

Szótárak

A szótárak és az állományok két típusa esetén történtek a léptetők terén a legnagyobb változások. A szótár típus léptetője a típus kulcsain lépked sorban végig. A `for mindenKulcs in myDict.keys()` utasítás lerövidíthető `for mindenKulcs in myDict` alakba, mint az az *1. listában* is látható.
Emellett három új, léptetéssel kapcsolatos tagfüggvény is bevezetésre került: `myDict.iterkeys()` – léptetés a szótár kulcsain, `myDict.itervalues()` – léptetés a szótár értékein és `myDict.iteritems()` – léptetés a kulcs/érték párokon.
Az `in` műveletjel úgy módosult, hogy a segítségével ellenőrizhető, hogy egy adott kulcs létezik-e a szótárban. Ez annyit tesz, hogy a `myDict.has_key(`*`valamilyen_kulcs`*`)` kifejezés *`valamilyen_kulcs`* `in myDict` alakra egyszerűsíthető.

Állományok

Az állományobjektumokhoz egy olyan léptetőobjektum társul, mely a `readline()` tagfüggvényt hívja meg. Vagyis az `in` műveletjellel egy szöveges fájl sorain lépkedhetünk végig – az eddig megszokott `for` *`egySor`* `in myFile.readlines()` alak helyett a sokkal egyszerűbb *`egySor`* `in myFile` alakban:

```
>>> myFile = open(·config-win.txt·)
>>> for egySor in myFile:
...     print egySor,
   # egy tovÆbbi œj sor beszœrÆsÆval az elızı
# sorzÆr  vesszı feladatÆnak megvÆltoztatÆsa
...
[EditorWindow]
font-name: courier new
font-size: 10
>>> myFile.close()
```

Osztályok

Saját léptetők létrehozása osztályok esetén is lehetséges. Ilyen módon elkerülhető a `__getitem__()` osztály tagfüggvény túlterhelése. A `__getitem__()` túlterhelésével az osztályhoz tartozó valamennyi elem bármely sorrend szerint kikérhető, ezt azonban egyes objektumok logikailag nem teszik lehetővé. Ha a `__getitem__()` helyett léptetőt használunk, egyértelműen megadhatjuk, hogy a felhasználó egy adott objektummal milyen műveleteket végezhet.
Létrehozhatjuk egy osztály saját léptetési rendszerét, ha olyan módon takarjuk el az osztály `__iter__()` tagfüggvényét, hogy az saját magát adja vissza, vagyis minden objektum saját magával tér vissza. Ezt követően pedig a `next()` tagfüggvényt kell eltakarni:

```
def __iter__(self):
    return self

def next(self):
    # visszatØrØs a k vetkezı elemmel,
# vagy StopIteration kivØtel dobÆsa.
```

Ezek alapján nézzünk meg egy egyszerű példát! Ezúttal egy sorszámozott listából térünk vissza egy véletlenszerűen kivá-

2. lista Osztályok léptetése

```
import random
class RandSeq:
    def __init__(self, seq):
        self.data = seq

    def __iter__(self):
        return self

    def __next__(self):
        return random.choice(self.data)

>>> from randseq import RandSeq
>>> for eachItem in RandSeq((·kı·, ·pap r·,
    ·oll ·)):
>>>     print eachItem
oll
oll
kı
pap r
pap r
oll
    :
```

lasztott elemmel (*2. lista*). Ez a példa a sorszámozott típusok léptetőinek néhány nem szabványos alkalmazását mutatja be. Az sem nevezhető szokványosnak, hogy esetünkben ez a léptetés végtelen. Mivel az elemeket szabálytalan sorrendben hívjuk meg, sosem futunk ki az elemekből, így a `StopIteration` kivétel sem jelentkezik.
A *3. listában* osztályunk felhasználásával egy léptetőobjektumot hozunk létre, de ahelyett, hogy egyszerre csak egy elemmel térnénk vissza, a `next()` tagfüggvényben értékként megadjuk, hogy a tagfüggvény egyszerre hány elemmel térjen vissza. Lássuk csak:

```
>>> a = AnyIter(range(10))
>>> i = iter(a)
>>> for j in range(1,5):
>>>     print j, ·:·, i.next(j)
1 : [0]
2 : [1, 2]
3 : [3, 4, 5]
4 : [6, 7, 8, 9]
```

Változtatható objektumok és léptetők

Mielőtt továbblépnénk az előállítókra, jegyezzük meg, hogy nem éppen okos dolog akkor megváltoztatni az objektumok tulajdonságait, amikor éppen valamilyen léptetővel dolgozunk rajtuk. Ez már a léptetők feltűnése előtt is nehézségeket okozott. Népszerű példa ennek bemutatására, ha egy listán próbálunk meg végiglépdelni és bizonyos feltételek teljesülése (vagy nem teljesülése) esetén egyes elemeket eltávolítani belőle:

```
for egyURL in osszesURL:
  if not egyURL.startswith(·http://·):
      mindenURL.remove(egyURL)        # IGEN!!
```

A listát kivéve a sorszámozott típusok mindegyike megváltoztathatatlan, így csak a listákkal lehet gond. Egy sorszámozott

3. lista Léptetőobjektum létrehozása saját osztállyal

```
#!/usr/bin/env python

class AnyIter:  # t bb elem lØptetØse
  def __init__(self, data, safe=0):
      self.safe = safe        # legy nk
                              #  vatosak
      self.iter = iter(data)  # a lØptetıvel

  def __iter__(self):         # lØpteti-
                              # osztÆly
      return self

  def next(self, howmany=1):  # k l nleges
                              # next()
      retval = []
      for eachItem in range(howmany):
          try:

retval.append(self.iter.next())
          except StopIteration:
              # kivØtel dobÆsa tœl
              # sok elem kØrØse esetØn
              if self.safe == 0:
                  raise
              # "biztonsÆgos"  zemm d:
              # a kØrtnØl kevesebb elem
              # visszaadÆsa
              else:
                  break
      return retval
```

típus esetén a léptető csak a éppen pillanatnyi elem sorszámát tárolja, így ha megváltoztatod a lista elemeit, az érintett elemeket a léptető figyelmen kívül hagyja. Ha kifutsz az elemekből, értelemszerűen `StopIteration` kivétel dobódik, amennyiben viszont új elemeket adsz a listához, a léptetés folytatható (*4. lista*).
Ha egy szótár kulcsai között léptetsz, közben a szótárt nem szabad megváltoztatnod! A szótár `keys()` tagfüggvénye által visszaadott lista természetesen kivétel ez alól, mert teljesen független a szótár tartalmától.
A léptetők ugyanakkor sokkal szorosabb kapcsolatban állnak a pillanatnyi objektummal, mint hinnénk, és nem hagyják, hogy az ilyen objektumokkal tudatlanul játszadozzunk:

```
>>> myDict = {·a·: 1, ·b·: 2, ·c·: 3, ·d·: 4}
>>> for egyKulcs in myDict:
...     print egyKulcs, myDict[egyKulcs]
...     del myDict[egyKulcs]
...
a 1
Traceback (most recent call last):
  File "", line 1, in ?
RuntimeError: dictionary changed size during
              ➥iteration
```

Ennek a figyelmességnek köszönhetően a hibát okozó szerkezetek egyszerűen elkerülhetők. A léptetőkkel kapcsolatos teljes leírásért forduljunk a 234-es PEP-hez.

4. lista Példa a léptetőkre

```
>>> langs = [·c·, ·perl·, ·tcl/tk·]
>>> langIter =iter(langs)
>>> langs.append(·c++·)
>>> langIter.next()
·c·
>>> langIter.next()
·perl·
>>> langIter.next()
·tcl/tk·
>>> langs
[·c·, ·perl·, ·tcl/tk·, ·c++·]
>>> langIter.next() # miutÆn a lØpteti
      # objektumot lØtrehoztuk
·c++·
>>> langs.remove(·perl·)
>>> langs.append(·ruby·)
>>> langs
[·c·, ·tcl/tk·, ·c++·, ·ruby·]
>>> langIter.next() # nincs t bb elem
Traceback (most recent call last):
  File "", line 1, in ?
    langIter.next()
StopIteration
>>> langs.append(·java·)   # œj elem
      # hozzÆadÆsa a sor vØgØhez
>>> langIter.next() # lØptetØs folytatÆsa
·java·
```

1. táblázat **Léptetők az előállítók ellenében**

Mi a `t`?	`t` megadása	A `t.next()` eredményeinek meghívása…
Iterator	`t = iter(obj)`	következő elem (vagy `StopIteration` kivétel)
Előállító	`def t(...):`	az eltárolt veremkeret visszaállítása… következő elem visszaadása … vagy `StopIteration` exception

Az előállítók

Az előállítók a léptetők alapötletéből fejlődtek ki. A fő érv az előállítók mellett az, hogy lehetővé teszik a függvények egyes hívásai közti állapotok elmentését. A statikus változók – a C függvényeiben megismertekhez hasonlóan – értékeiket képesek megőrizni a függvények egyes hívásai között. Ez részben megoldja a gondot, de igazi megoldást egy olyan változó bevezetése jelentene, amely léptetőként képes működni, de két függvényhívás közt is megőrzi az állapotát, és ha újra meghívják, ugyanott tudja folytatni, ahol előzőleg abbahagyta. Nos ez az, amit az előállítók valójában tesznek. Megvalósítják és ugyanakkor ötvözik a léptetők és a folytatható függvények működését. Ha meghívjuk őket, bármikor ugyanott bírják folytatni, ahol a futásuk előzőleg félbemaradt, mivel a folytatáshoz szükséges összes adatot tárolják, így bármikor képesek a következő elemet szolgáltatni. A feladat megvalósításához egy új kulcsszó is bevezetésre került, a `yield`, amely nem valódi függvényvisszatérést valósít meg, hanem csak egy értéket ad át (a keretobjektumot leemeli a veremről).

A visszirányú együttműködés biztosításáért (arra az esetre, ha egy már korábban létező program felhasználja a `yield` kulcsszót mint azonosítót) az előállítók használatát a `from __future__ import generators` utasítással előre kell jelezni. A létrehozók előbb-utóbb mindenképpen szabvánnyá válnak (2.3?), így erre a későbbiekben már nem lesz szükség. A létrehozóknak létezik még egy a léptetőkhöz hasonló tulajdonsága: ha a vezérlés eléri a függvény végét vagy egy `return` utasítást, és már nincs több átadandó érték, a léptetőkhöz hasonlóan az előállító is `StopIteration` kivételt dob. Egy egyszerű példa:

```
def egyszeruGen():
    yield 1
    yield ·2 --> punch!·
```

Most pedig hívjuk meg ezt a függvényt, és egy előállító objektumot fogunk visszakapni:

```
>>> egyszG = egyszeruGen()
>>> egyszG.next()
1
>>> egyszG.next()
·2 --> punch!·
>>> egyszG.next()
Traceback (most recent call last):
  File "", line 1, in ?
    egyszG.next()
StopIteration
```

Esetleg még találóbban: `for` *`mindenElem`* `in` *`egyszeru`*`Gen():print` *`mindenElem`*. Természetesen ez egy buta példa. Úgy értem, miért nem használunk inkább léptetőt erre a célra? Egyszerűbben megérthető, hogy mire gondolok, ha egy függvény rugalmasságát igénylő sorszámozott típuson próbálunk meg léptetni ahelyett, hogy ugyanezt egyszerű statikus objektumokat tartalmazó sorszámozott listával tennénk. A következő példában olyan véletlenszerű léptetőt hozunk létre, amely egy sorszámozott típust vár értékként, és annak egy véletlenszerűen kiválasztott elemével tér vissza:

```
from random import randint
def randIter(seq):
    while len(seq) > 0:
        yield seq.pop(randint(0, len(seq)-1))
```

Itt az a különbség, hogy minden elem, amellyel visszatérünk, egyben el is távolítódik, vagyis mintha a `list.pop()` és `random.choice()` függvényeket egyesítenénk:

```
>>> for mindenElem in randIter([123, ·xyz·,
    45.678, 9j]):
...     print mindenElem
...

·xyz·
9j
45.678
123
```

Az *1. táblázat* a léptetők és az előállítók közötti különbségeket összegzi. Ha fellapozod a 234-es és 255-ös PEP-eket, a témával kapcsolatos további érdekességeknek is utánanézhetsz.

Az osztásjel megváltozott tulajdonságai

Valószínűleg ennek a tulajdonságnak a léte a legvitatottabb az új Python-változatban. Természetesen számos érvet sorakoztathatunk fel ennek az új tulajdonságnak a megléte mellett, csakhogy ellene is akad egy-kettő. Végül azok győztek, akik valódi osztást szeretnének megvalósítani. Ahhoz, hogy megértsük, mit is jelent ez a változás valójában, hozzunk létre (újra) néhány osztási alapszabályt, és határozzuk meg, hogyan működjön egész és valós számok esetén.

- Klasszikus osztás
  Ha egész számokat osztunk, úgy klasszikus osztás esetén az eredmény is egész szám lesz, amely úgy jön létre, hogy a tizedesrészt az osztás levágja (lásd alább az Osztás kerekítéssel című részt). Ha valós számokat osztunk, az eredmény is valós szám lesz, mégpedig az osztás valódi végeredménye (lásd a Valódi osztás részt). Lássuk csak, hogyan működik az osztás napjainkban a Pythonban:

```
>>> 1 / 2    # egØsz szÆmok osztÆsakor
      # egØsz ØrtØket kapunk
0
>>> 1.0 / 2.0       # val s szÆmoknÆl
      # az eredmØny is val s
0.5
```

- Valós osztás
  Valós osztásról akkor beszélünk, ha egy osztás eredménye a két szám valódi hányadosát adja vissza. A Python 3.0 közeledtével ezen a téren is nagy változásoknak nézhetünk elébe. Jelenleg, ha valaki a valódi osztás lehetőségeit szeretné kihasználni, ki kell adnia a `from __future__ import division` utasítást. Ennek következtében az osztásjel valódi osztást fog megvalósítani:

```
>>> from __future__ import division
>>>
>>> 1 / 2    # val s eredmØnyt ad vissza
0.5
>>> 1.0 / 2.0       # mint ahogyan ez is
0.5
```

- Osztás kerekítéssel
  Az új osztásjel (`//`) minden esetben egy lefelé kerekített (helyesebben: a számegyenesen balra eltolt) egész számmal tér vissza, függetlenül a műveletben résztvevő értékek numerikus típusától. Ez a műveleti jel már a 2.2.-es változatban is megtalálható anélkül, hogy a `__future__`-ből bármilyen tulajdonságot be kellene illesztenünk:

```
>>> 1 // 2       # EgØsz szÆmok esetØn lefelØ
                 # kerek tett egØsz eredmØny.
0
>>> 1.0 // 2.0   # Val s szÆmok esetØn lefelØ
                 # kerek tett val s eredmØny.
0.0
>>> -1 // 2      # Ennek az eredmØnye egy a
                 # szÆmegyenesen balra eltolt
                 # ØrtØk.
-1
```

Anélkül, hogy bármiféle vitába bocsátkoznánk ezzel a változással kapcsolatban, az érzés olyan, mintha a Python osztásjele a kezdetek óta hibás lenne, különösen azok számára, akiknek a Python az első programozási nyelvük, és nincsenek hozzászokva az effajta osztáshoz. Ezt ***Guido van Rossum*** a What's New in Python 2.2 (A Python 2.2 újdonságai) című ZPUG-beszélgetésben a következőképpen szemlélteti:

```
def sebesseg(tavolsag, osszesIdo):
    arany = tavolsag / osszesIdo
```

Ez a példafeladat rossz, mivel az eredménye teljes mértékben a felhasznált numerikus típusoktól függ. Ha egész számokat adsz meg eredménynek, merőben más eredményhez jutsz, mintha valós típusokat használnál. Hogy eloszlasd a kétséget, a következő átalakításokat kell fejben elvégezned:

```
>>> 1 == 1.0
1
>>> 2 == 2.0
1
>>> 1 / 2 == 1.0 / 2.0    # klasszikus osztÆs
0
```

Ha a Python újfajta osztását használod fel, ismét helyreáll a rend a világegyetemben:

```
>>> from __future__ import division
>>> 1 / 2 == 1.0 / 2.0    # val di osztÆs
1
>>> 1 // 2 == 1.0 // 2.0  # kerek tØses
                          # osztÆs
1
```

Mindamellett, hogy az effajta megoldás tűnik igazinak és helyesnek, alkalmazása esetén sem kerülhetjük el korábbi kódjaink hibássá alakulását. Szerencsére ennek a Python fejlesztői is tudatában voltak, és ez az újfajta osztási tulajdonság csak a 3.0-s változattól kezdődően lesz az alapértelmezett, ami még évekre van tőlünk. Azok, akik ezt az új osztási formát kívánják felhasználni, ezt a tulajdonságot beilleszthetik, vagy a Pythont a `-Qnew` kapcsolóval meghívva az újfajta osztást ugyancsak érvényre juttathatják. Létezik néhány kapcsoló, melyeket megadva a Python figyelmeztet, ha a régi osztás tulajdonságaira építünk, így készítve fel minket a majdani Python 3.0-s időkre. A 238-as PEP-ből további adatokat tudhatsz meg, és a témával kapcsolatban a ➲ http://comp.lang.python csoportban folyt heves vitákat sem árt átolvasgatni. A 2. *táblázat* összegzi az osztással kapcsolatos új és régi tulajdonságokat.

2. táblázat **Az osztásjel új tulajdonságai**

Műveleti jel	2.1.x vagy régebbi	2.2 vagy újabb („import division" nélkül)	2.2 vagy újabb („import division" megadásával)
/	klasszikus	klasszikus	valódi
//	–	kerekítéses	kerekítéses

Típusok és osztályok összefésülése

A típusok és osztályok összefésülése már jó ideje a Python kívánságlistáján várakozott. A programozók csalódottak voltak, hogy a már létező típusokat (például: a listákat) nem tudták

továbbszármaztatni, sem a tulajdonságaikat kibővíteni.
A témával kapcsolatban ajánlatos mind a kapcsolódó PEP-eket, mind Guido írását fellapozni, ami azok számára készült, akik gyorsan meg szeretnének ismerkedni ezzel az új tulajdonsággal, anélkül, hogy a PEP-eken kellene átrágni magukat, és esetleg elveszniük az ott felsorakoztatott megannyi részletben (lásd a ***Kapcsolódó címeket***). Mindezek szemléltetésére egy ötletes példát is láthatunk, melyben a Python listáit bővítjük ki fejlett veremkezelési lehetőségekkel.
A stack2.py példaprogram megszületését egy korábbi léptetős példa motiválta (lásd a 6.2-es példát a Core Python Programming oldalon).

```
#!/bin/env python

·stack2.py   szÆrmaztat Øs kibıv t egy listÆt·

class Stack(list):

  def __init__(self, *args):
      list.__init__(self, args)
      # a sz lı osztÆly
      # lØtrehoz  f ggvØnyØnek megh vÆsa

  def push(self, *args):
      for eachItem in args:
      # egyszerre t bb elem
          self.append(eachItem)
      # is tÆrolhat

  def pop(self, n=1):
      if n == 1:   #  egyetlen elem kiemelØse
           return list.pop(self)
      else:  # t bb elem
      # visszakØrØse
          return [ list.pop(self) for i in
          ➥range(n) ]
```

Alább látható a legutóbbi példánk kimenete:

```
>>> from stack2 import Stack
>>> m = Stack(123, ·xyz·)
>>> m
[123, ·xyz·]
>>> m.push(4.5)
>>> m
[123, ·xyz·, 4.5]
>>> m.push(1+2j, ·abc·)
>>> m
[123, ·xyz·, 4.5, (1+2j), ·abc·]
>>> m.pop()
·abc·
>>> m.pop(3)
[(1+2j), 4.5, ·xyz·]
>>> m
[123]
```

Azon túl, hogy képesek vagyunk beépített típusokat származtatni, érdemes még kiemelni az új stílusú osztályok egyes tulajdonságait:

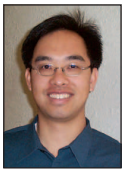
- Osztálylétrehozó függvények típuskényszerítése
  – Új `__class__`, `__dict__` és `__bases__` tulajdonságok.
  – A `__getattribute__()` különleges tagfüggvény (a `__getattr__()` okosabb fajtája).
- Osztályleírók
  – Osztálytulajdonságok.
  – Statikus tagfüggvények.
  – Osztály tagfüggvények.
  – A szülő tagfüggvényeinek meghívása.
  – Együttműködő tagfüggvények.
  – Új gyémántdiagram nevű feloldás (new diamond diagram name resolution).
  – Korlátozott készletű osztálytulajdonságok gyenge kötésekkel.

Az új stílusú osztályokkal és a típusok és osztályok egyesítésével kapcsolatos tájékoztatásért fordulj a 252-es és 253-as PEP-ekhez vagy a korábban említett Guido-íráshoz.

Összegzés

Amellett, hogy a Python új tulajdonságai egy csomó korábbi hibát orvosolnak, és nagymértékben előrelendítik ezt az eszközt, néhányan úgy gondolják, hogy ezek már sértik a Python korábbi egyszerű mivoltát. Ha szigorúan nézzük a dolgokat, talán helyt adhatunk ennek a kétségnek. Mégis azáltal, hogy eltüntetjük a korábbi zavaró hibákat és új tulajdonságokkal bővítjük a Pythont, valószínűleg egy sokkal használhatóbb eszközhöz jutunk. Az új változások semmilyen hatással nincsenek a korábbi programokra, vagy amennyiben mégis – mint az új osztásjel esetén –, úgy az alkalmazkodásra és az átállásra még jócskán van időnk.
Végezetül pedig a ***Kapcsolódó címekben*** néhány komolyabb leírást találhatunk, úgymint ***Andrew Kuchlin*** „A Python 2.2 újdonságai” című írását, valamint egy előadást Guidótól, amelyet a még ősszel megrendezett Python-találkozón tartott.
A Python 2.2.1-es változata letölthető a Python honlapjáról.
Jó munkát mindenkinek!

Linux Journal 2002. július, 99. szám



Wesley J. Chun
a „Core Python Programming” írója. Több mint egy évtizede foglalkozik programozással és oktatással. A Yahoo! „Mail” és „People” szolgáltatásai is részben az ő segítségével jöttek létre. Jelenleg a Synarcnál, egy klinikai szolgáltatásokkal foglalkozó cégnél dolgozik, ahol Pythonban fejlesztett alkalmazások segítségével a betegek panaszait elemezve próbálják megkönnyíteni a radiológián dolgozók munkáját.

Kapcsolódó címek

Dr. David Mertz – Charming Python: Iterators and simple generators (A Python léptetőiről és előállítóiról), 2001. szeptember.
➲ http://www-106.ibm.com/developerworks/library/l-pycon.html?n-l-9271
Wesley J. Chun – Core Python Programming a Prentice Hall kiadó gondozásában, 2001.
Core Python Programming kiegészítő honlap
➲ http://starship.python.net/crew/wesc/cpp