Python oktató

Release 2.4.1

Guido van Rossum, Fred L. Drake, Jr., editor

Utolsó frissítés: 2006. február 26.

Python Software Foundation
Email: docs@python.org

.

A fordításhoz észrevételek:
E-mail: diogenesz@pergamen.hu

Pár szó a magyar fordításról

Ez a fordítás Horváth Árpád Python tutorial fordításának a folytatása és aktualizált verziója. A fordítás a 2.1-es verziójú tutoriallal kezdődött, ezt a verziót teljes egészében a 2.4.1-es verzióhoz igazítottam.

A tutorial fordítását Horváth Árpád kezdte el, én az ő munkáját folytatom.
Árpád elérhetősége: `ahorvath at szgti.kando.hu`
(at helyett @ kell, és egybeírni, így talán kevesebb spam-et kap)

A tutorialt jelenleg gondozza: Nyírő Balázs (`diogenesz at pergamen.hu` – ha van valamilyen észrevételed a fordítással kapcsolatban, írd meg!)

Ha fordítani szeretnél (bármilyen dokumentációt) ...

Pár megjegyzés és segítség a fordításhoz

Nem kell megijedni, nem nehéz LaTeX-ben lévő szöveget írni, egyszerű szövegszerkesztő (notepad, emacs, joe, vim...) kell hozzá, és segítek ha kell. Túl sok ismeret nem szükséges a LaTeX-ből.

Hogy könnyebben ellenőrizhető legyen, a fordításnál érdemes benne hagyni a forrásban az angol szöveget megjegyzésbe téve – % a sor elején, példa itt:

```
% Finally, the least frequently used option is to specify that a
% function can be called with an arbitrary number of arguments.  These
% arguments will be wrapped up in a tuple.  Before the variable number
% of arguments, zero or more normal arguments may occur.

ez már nincs megjegyzésbe téve
```

A LaTeX-ben üres sor jelöli, hogy új bekezdés jön.

A LaTeX-ben találkozhatsz utasításokkal és környezetekkel. Azokat hagyd ugyanúgy. Az alábbi példában a `code` egy utasítás, a `print` az argumentuma. (A kapcsos zárójel kötelező, a szögletes opcionális argumentumot jelöl.) Az `item` is egy utasítás. Az `itemize` egy környezet (rendezetlen (sorszámozatlan) listát készíthetsz vele).

```
A \code{print} utasítással ...

\begin{itemize}
\item
a magasszintű adattípusok lehetővé teszik számodra egyetlen utasításban
egy összetett művelet kifejtését;
\item
az utasítások csoportosítása a sorok elejének behúzásával történik
a kezdő és végzárójelek helyett;
\item
nem szükséges a változók és argumentumok deklarálása.
\end{itemize}
```

A kritikus részeket tripla kérdőjellel (???) jelöljük meg. Itt azokra a helyekre gondolok, amit másnak érdemes átnéznie, hogy jó-e úgy, ami érthetőség, pontosság, magyar nyelv szempontjából valószínűleg javítandó.

A Python dokumentációkészítésről a http://www.python.org oldalon bővebb leírást találhatsz. Fent van egy link: Documentation » Download Current Documentation (középtájt) » Documenting Python (lent). (url: http://docs.python.org/doc/doc.html) A Python dokumentáció egyben is letölthető ugyanitt html, pdf formátumban.

Ha Linuxod van, és van fent LaTeX (`tetex-*` csomagok), és latex2html (`latex2html` csomag), akkor megpróbálhatod lefordítani a forrást html-lé.

Tapasztalatok: a fordítást a teljes Python forrással együtt érdemes letölteni, ekkor működött nekem rendesen a beépített html és pdf generálási funkció. A Python dokumentáció előre elkészített generátorral rendelkezik, ami a LaTeX fájl alapján a html-t és a pdf-et automatikusan generálja.

Angol szavak/kifejezések magyar megfelelői

Gondban vagyok a ´string literal' szóval.

A 'prompt' szóra sem tudtam jó magyar megfelelőt, hagytam az eredetit.

A backslash jelölését a LaTeX könyv ötletes rep-jel megnevezésével fordítottam először. Újabb tanács alapján kicseréltem az elterjedtebb vissza-per jel megnevezésre.

A tuple szóra eddig a vektor szót használtuk – hosszútávon az eredeti tuple kifejezést fogjuk visszaállítani.

Várom a javaslatokat. Szívesen veszek minden ötletet, ami a magyarosabbá-tételt segíti.

Szótár

angol	**magyar**
argument	argumentum (?paraméter)
built-in function	beépített/belső/alap függvény
control flow statements	vezérlő utasítások
string-literal	???
backslash	vissza-per jel (rep-jel)
prompt	prompt???
tuple	tuple
method	eljárás <- metódus, javítás alatt
sequence	sorozat
object	objektum
exceptions	kivétel
raise an exception	kivételt vált ki
clause	mellékág
handler (az exeption-öknél)	kezelő???
token	token???
default	alapértelmezett
scope	hatáskör, hatókör, érvényességi kör
keyword argument	kulcsszavas argumentum
right-hand side expression	???
sequence unpacking	a sorozat szétbontása???
shortcut operators	???
slice notation	szelet jelölési mód???
List Comprehensions	???lista értelmezés
Library reference	Referencia könyvtár

Kivonat

A Python egy könnyen tanulható, sokoldalú programozási nyelv. Jól használható magas szintű adatstruktúrákkal rendelkezik, és egyszerű – ugyanakkor eredményes az objektum-orientált programozás nyelvi megvalósítása. A Python nyelvtana elegáns, a nyelv dinamikusan típusos. Parancsértemlező jellegű nyelv, mely a bájtfordítást is támogatja (mint pl. a Java), ideális szkriptek készítésére, továbbá nagyon gyors alkalmazásfejlesztést tesz lehetővé több platfromon, szerteágazó területeken.

A Python értelmező és a sokrétű, alaposan kidolgozott alap-könyvtár (Standard Library) szabadon elérhető és felhasználható akár forráskódként, akár bináris formában minden jelentősebb platformra a Python weboldaláról: http://www.python.org/. Ugyanitt hivatkozások is vannak külső fejlesztésű modulokra, programokra és eszközökre, és kiegészítő dokumentációkra.

A Python értelmező egyszerűen kiegészíthető C vagy C++ nyelven (vagy más, C-ből hívható nyelven) új függvényekkel és adattípusokkal. A Python maga is alkalmas kiegészítő nyelv már elkészült alkalmazások bővítéséhez.

Ez az oktató megismerteti az olvasóval a Python alapvető gondolkodásmódját, nyelvi lehetőségeit és rendszerét. Segít a gyakorlott felhasználóvá válásban. A példák a dokumentumba ágyazottak, internet elérés nem szükséges az olvasásukhoz.

A nyelvbe épített szabványos objektumok és modulok leírásához nézd meg a *Python Library Reference (Python szabványos könyvtár)* készülő magyar fordítását (és ha van kedved, segíts a fordításban!).

A *Python Reference Manual* a nyelv részletes definícióját tartalmazza. C vagy C++kiegészítések írásához hasznos olvasnivaló az *Extending and Embedding the Python Interpreter* és a *Python/C API Reference*. Ezenkívül van néhány könyv, amely a Pythonnal komolyan foglalkozik.

Ez nem egy minden apró részletre kiterjedő oktatóanyag. Bemutatja a Python legfontosabb lehetőségeit, és megismerteti veled a nyelv gondolkodásmódját és stílusát. Ha végigcsinálod, képes leszel Python modulok és programok írására, és ami a legfontosabb: a további tanulásra. Az oktató után érdemes a *Python Library Reference (Python szabványos könyvtár)*-ban leírt modulokkal megismerkedni.

TARTALOMJEGYZÉK

1. fejezet

Étvágygerjesztő

Ha valaha is írtál hosszú shell szkriptet, feltehetően ismered azt az érzést, hogy amikor új alaptulajdonságot szeretnél hozzáadni, a program lassúvá és bonyolulttá válik; vagy az új tulajdonság magában foglal egy rendszerhívást vagy más függvényt, amely csak C programból érhető el... Gyakran a probléma nem olyan komoly, hogy a C-ben történő újraírást indokolná; talán a programnak szüksége van változó hosszúságú karakterláncokra vagy más adattípusra (mint amilyen a fájlnevek rendezett listája), melyet könnyű létrehozni a shell-ben, de rengeteg munka C-ben, vagy talán nem ismered eléggé a C nyelvet.

Más helyzet: talán több különböző C könyvtárakkal kell dolgoznod, és a gyakori írás/fordítás/tesztelés/újrafordítás ciklus túl lassú. Sokkal gyorsabban kellene szoftvert fejlesztened. Talán írtál egy programot amely képes kiegészítő nyelv használatára – és te nem akarsz csak emiatt tervezni egy nyelvet, írni és javítgatni egy fordítót (interpretert).

Ha valamelyik állítás igaz rád, a Python megfelelő nyelv lehet számodra. A Pythont egyszerű kezelni, mégis igazi programnyelv, sokkal több szerkezetet használ és több támogatást nyújt nagyméretű programok számára, mint a shell. Ezzel egyidőben sokkal több hibaellenőrzési lehetőséget tartalmaz mint a C, és – lévén *magas szintű nyelv* – magas szintű beépített adattípusai vannak, úgymint rugalmasan méretezhető sorozatok és szótárak, amelyeket C-ben létrehozni napokba tellene. Az általánosabban megfogalmazott adattípusaival a Python jóval nagyobb problématerületen alkalmazható mint az *Awk* vagy akár a *Perl*, ugyanakkor sok dolog legalább ugyanolyan könnyű Pythonban, mint ezekben a nyelvekben.

A Python lehetővé teszi, hogy a programodat modulokra oszd fel, amelyek felhasználhatók más Python programokban is. A nyelvhez tartozó alap-könyvtár alaposan kidolgozott modulgyűjteményt tartalmaz, melyeket a programod alapjául használhatsz – vagy példáknak a Python tanulásához. Vannak beépített modulok is mint a fájl I/O, rendszerhívások, socket-ek kezelése és interfészek olyan grafikus felülethez, mint a Tk.

A Python futási idő alatt értelmezett (interpretált) nyelv, amely időt takarít meg neked a programfejlesztés alatt, mivel nincs szükség gépi kódra történő fordításra és a gépi kódok összeszerkesztésére. Az értelmező interaktívan is használható, lehet kísérletezni a nyelv tulajdonságaival, vagy függvényeket tesztelni az alulról felfelé történő programfejlesztés során. Egyben egy ügyes asztali számológép is!

A Python nagyon tömör és olvasható programok írását teszi lehetővé. A Pythonban írt programok általában sokkal rövidebbek mint a C vagy C++ megfelelőjük, mert:

- a magasszintű adattípusok lehetővé teszik egyetlen utasításban egy összetett művelet kifejtését;
- az utasítások csoportosítása a sorok elejének egyenlő mértékű jobbra tolásával történik a kezdő és végzárójelek helyett;
- nem szükséges a változók és argumentumok deklarálása.

A Python *bővíthető*: ha tudsz C-ben programozni, akkor könnyű új beépített függvényt vagy modult hozzáadni az értelmezőhöz, vagy azért, hogy hogy a kritikus eljárások a lehető leggyorsabban fussanak, vagy például olyan könyvtárakra linkelni Pythonból, amelyek csak bináris formában érhetők el (amilyenek a forgalmazóspecifikus grafikai programok). Hogyha a nyelv valóban mélyen megfogott, akkor a Python értelmezőt hozzákötheted egy C-ben írt alkalmazáshoz, és azt a program kiterjesztéseként vagy parancs-nyelvként használhatod.

A nyelv a „Monthy Python-ék Repülő Cirkusza” nevű BBC-s sóműsor után kapta a nevét és semmi köze nincs a nyálas hüllőhöz... Utalásokat tenni a dokumentációban a Monty Pythonra nemcsak szabad, hanem ajánlott!

Most, hogy már felkeltette az érdeklődésedet a Python, reméljük szeretnéd megtekinteni valamivel részletesebben. Mivel a nyelvtanulás legjobb módja annak használata, meghívunk Téged egy kis gyakorlásra.

A következő fejezetben az értelmező használatának minkéntjét magyarázzuk. Ezek kissé unalmas dolgok, de szükségesek ahhoz, hogy a későbbiekeben mutatott példákat kipróbálhasd.

Az oktató többi része példákon keresztül mutatja be a Python nyelv és a rendszer sok-sok tulajdonságát, kezdve az egyszerű kifejezésekkel, utasításokkal és adattípusokkal – folytatva a függvényekkel és a modulokkal. Végül érinti a legújabb programozási módszereket, mint például a kivételkezelés, és a felhasználó által definiált osztályok.

2. fejezet

A Python értelmező használata

2.1. Az értelmező elindítása

The Python interpreter is usually installed as '/usr/local/bin/python' on those machines where it is available; putting '/usr/local/bin' in your UNIX shell's search path makes it possible to start it by typing the command

A Python értlemező szokásos helye a '/usr/local/bin/python' könyvtár, ahol ilyen létezik; helyezd el a '/usr/local/bin' útvonalat a UNIX shell keresési útvonalán, és ekkor a

```
python
```

utasítás beírásával az értelmező elindítható.

Mivel a telepítés helye telepítési opció, más helyen is lehet a program; ha nem tudod az értelmezőt elindítani, kérj segítséget egy nagyobb tudású embertől. (Például a '/usr/local/python' is egy gyakori hely.)

Fájlvége karaktert (`Control-D` UNIX-on, `Control-Z` Windows-on) írva a Python elsődleges promptjára, az értelmező nulla kilépési státusszal (zero exit status) lép ki. Ha ez nem működik, akkor az értelmezőt a következő utasításokkal hagyhatod el: '`import sys; sys.exit()`'.

A szövegsori szerkesztés támogatottságának talán leggyorsabb ellenőrzése a Control-P megnyomása az első megjelenő promptnál. Ha ez sípol, akkor van; lásd az Appendix bemutatja az érvényes billentyűket. Ha semmi nem jelenik meg, vagy csak egy `^P`, akkor a szövegsori szerkesztés nem elérhető; ekkor csak a backspace karakterrel törölhetsz karaktert az aktuális sorból.

Az értelmező a UNIX shellhez hasonlóan működik: ha a szabváyos bemenettel van meghívva, akkor interaktívan olvas és hajt végre utasításokat; ha fájlnév argumentummal vagy egy fájllal, mint szabványos bemenettel hívjuk meg, akkor *szkript*-ként lefuttatja a megadott fájlt.

Egy másik lehetőség az értelmező indítására a '**python -c** *command* `[arg] ...`' parancsor, amely végrehajtja az utasítás(oka)t a *command*-ban, hasonlóan a shell **-c** opciójához. Mivel a Python utasítások gyakran tartalmaznak szóközöket, vagy más speciális karaktereket, amelyeknek speciális jelentésük van a shell számára, jobb, ha a *parancs* részt idézőjelbe (double quote) tesszük.

Some Python modules are also useful as scripts. These can be invoked using '**python -m** *module* `[arg] ...`', which executes the source file for *module* as if you had spelled out its full name on the command line.

Némelyik Python modul önálló programként is használható, ha így hívod meg: '**python -m** *module* `[arg] ...`' – ez az utasítás végrehajtja a *module* forráskódját.

Érdemes megjegyezni, hogy különbség van a '`python file`' és a '`python <file`' között. A második esetben a program beviteli kéréseit (amilyenek a `input()` és `raw_input()` hívások) a program a *fájlból* veszi. Mivel a fájlt az elemző (parser) a fájl végéig beolvassa a program indulása előtt, a program azonnal fájl-vége karakterrel fog találkozni. Az első esetben (amely gyakran a kívánt működés) a program azokat a fájlokat vagy eszközöket (device) használja, amelyek a Python értlemezőhöz csatlakoznak.

Sokszor hasznos, ha egy szkript fájl futtatása után rögtön interaktív üzemmódba kerülünk. Ezt egyszerű elérni, a szkript neve előtt adjuk meg az **-i** kapcsolót. (Ez nem működik ha a szkriptet a szabványos bemeneten keresztül olvassuk be, a fentebb leírtak szerint.)

2.1.1. Argumentum átadás

Ha az értelmezőt szkrip-fájllal indítjuk, akkor a szkript fájl neve és a nevet esetleg követő argumentumok a `sys.argv`, string-lista típusú változóba kerülnek. Ennek hossza leglább 1; amennyiben sem szkrip-nevet, sem semmilyen argumentumot nem adunk meg, a `sys.argv[0]` változó értéke üres string lesz. Ha a szkript neveként `'-'`-t adunk meg (ez a szabványos bemenetnek felel meg), `sys.argv[0]` értéke `'-'` lesz. Argumentumként **-c** *parancs*-ot megadva `sys.argv[0]` értéke `'-c'` lesz. A **-c** *parancs* után álló kapcsolókat az értelmező nem dolgozza fel, hanem a `sys.argv`-ben eltárolva a *parancs*-ra hagyja annak feldolgozását.

2.1.2. Interaktív (párbeszédes) mód

Amikor a parancs a tty-ről (pl. billentyűzet) érkezik, az értelmező úgynevezett *interaktív* (párbeszédes) üzemmódban működik. Ekkor az *elsődleges prompt* megjelenítésével kéri a következő parancs megadását, amely szokásosan három egymás után következő nagyobb-jel (‘`>>> `’); a sorok folytatásához ezek a folytatólagos sorok – a *másodlagos promptot* jeleníti meg, szokásos értéke három egymás után írt pont (‘`... `’). Azt értelmező elindulásakor, mielőtt az elsődleges prompt megjelenne, egy üdvözlő-szöveget ír ki, tartalmazva az értelmező verziószámát és a jogi védettséget

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Folytatólagos sorok szükségesek, ha többsoros szerkezeteket írsz be. Például a következő `if` utasítás esetén:

```
>>> a_vilag_sik = 1
>>> if a_vilag_sik:
...     print "Vigyázz, nehogy leess!"
...
Vigyázz, nehogy leess!
```

2.2. Az értelmező és környezete

2.2.1. Hibakezelés

Hiba esetén az értelmező hibaüzenetet küld és kiírja a hiba nyomvonalát (stack trace). Párbeszédes üzemmódban az elsődleges promt jelenik meg; ha az adatok beolvasása fájlból történt, kiírja a hibanyomvonalat és nullától eltérő kilépési értékkel tér vissza. (A programon belül fellépő és a program által kezelt kivételek (`except - try`) nem jelentenek hibát ebben a környezetben.) Vannak feltétel nélküli, ú.n. fatális hibák, amelyek azonnal, nullától eltérő visszatérési értékkel történő programmegszakítást és kilépést eredményeznek, ilyenek a belső inkonzisztenciát okozó, valamint a memóriából kifutó programok hibái. Minden hibaüzenet a szabványos hibakimenetre (`standard error`) kerül, a futtatott parancsok rendes kimenete pedig a szabványos kimenetre (`standard output`) íródik.

Ha az elsődleges, vagy a másodlagos promptnál a megszakítás-karaktert gépeljük be (rendszerint Control-C vagy a DEL billentyű), az törli az eddigi bemenetet és visszaadja a vezérlést az elsődleges promtnak. [1]

[1] A GNU Readline csomag hibája ennek a funkciónak a működését megakadályozhatja.

Ha a megszakításkérelmet valamely parancs/program végrehajtása alatt adjuk ki, az `KeyboardInterrupt` kivételt generál; ez programból a `try` utasítással 'kapható el'.

2.2.2. Végrehajtható Python szkriptek

A BSD-szerű UNIX rendszereken (Linuxon is) a Python szkripteket ugyanúgy, mint a shell szkripteket - közvetlenül futtathatóvá lehet tenni, legelső sorként megadva a

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

(feltételezve, hogy a Python értelmező elérési útvonala a felhasználó PATH változójában be van állítva és a szkript fájl végrehajtható tulajdonsággal bír).

Ford. megjegyzés: Debian rendszeren a Python elérési útvonalát adtam meg a programjaim elején, ez nálam

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

volt – így nem volt szükség környezeti változó beállítására.

A fenti esetekben a fájlnak a '`#!`' karakterekkel kell kezdődnie. Ezt a sort néhány platformon UNIX-típusú sorvéggel ('`\n`') kell lezárni, nem pedig Mac OS ('`\r`') vagy windows-os ('`\r\n`') sorvéggel.

Figyelem: ha láthatatlan '`\r`' karaktert tartalmaz az első sor vége, nem fog lefutni a program, és a hibát nagyon nehezen lehet észrevenni! Tipikusan akkor fordul elő, ha Windows-os szövegszerkesztővel (pl. notepad) készítjük a programot.

Megjegyezendő: a '#' karakterrel a Pythonban a megjegyzéseket kezdjük.

A szkriptnek a **chmod** paranccsal adhatunk végrehajtási engedélyt az alábbi módon:

```
$ chmod +x szkriptem.py
```

2.2.3. A forráskód karakterkészlete

A Python-os forrásfájlokban az ASCII (7-bites) karakterkódolástól eltérő kódlapokat (karakterkészletek) is használhatunk. Legegyszerűbben a `#!`-vel kezdődő sort követően az alábbi módon adhatjuk meg a használt karakterkészletet/kódtáblát:

```
# -*- coding: iso-8859-1 -*-
```

Ezt követően a forráskód minden karaktere az `iso-8859-1` karaktertáblázat alapján értelmeződik, ezáltal közvetlenül unikódos (Unicode) beégetett szövegrészeket (string literal) használhatunk a programunkban. A választható kódlapok/karakterkészletek listája a *Python Library Reference* -ben a codecs fejezetben található.

For example, to write Unicode literals including the Euro currency symbol, the ISO-8859-15 encoding can be used, with the Euro symbol having the ordinal value 164. This script will print the value 8364 (the Unicode codepoint corresponding to the Euro symbol) and then exit:

Például olyan Unicode karaktereket akarsz írni, amelyek tartalmazzák az Euro pénznem jelét, az ISO-8859-15-ös kódolást használhatod – melyben az Euro a 164-es karakterhelyen található.

A következő kis program Windows XP-n tesztelve 128-as értéket ír ki. (az előbbi 164-es értéket nem tudom miért írták a tutorialba, és miért 128-at ad vissza nálam a példaprogram - ford.)

```
# -*- coding: iso-8859-15 -*-

currency = u"■" # itt az euro szimbólumot gépeltük be.
print ord(currency) # kimenete XP-n 128
print chr(128) # euro szimbólumot ír ki.
```

Amennyiben a szövegszerkesztőd lehetővé teszi a fájl `UTF-8`-ként való elmentését byte-sorrend megjelöléssel (BOM), akkor elég így elmenteni a fájlt és nem kell a fenti deklaráció. A fejlesztőkörnyezet (IDLE) ezt az `Options/General/Default Source Encoding/UTF-8` menü útvonalon keresztüli beállítással támogatja. Megjegyzendő, hogy ezt a tulajdonságot csak a Python 2.3-as és a fejlettebb verziók kezelik helyesen; az operációs rendszer nem kezeli le, és ebben az esetben UNIX rendszereken nem is tudja a `#!` sorozattal kezdődő (parancs)fájlokat értelmezni!

UTF-8 használatával (vagy a BOM, vagy a kódolás program eleji megadásával) beégetett szövegek (string literals) és a megjegyzések a világ legtöbb nyelvén beírhatók a programba – egy fájlban több nyelvet is használhatunk. A változók neveiben továbbra is csak ASCII karaktereket szabad használni. A szövegszerkesztő csak akkor jeleníti meg hibátlanul a karaktereket, ha felismeri hogy UTF-8 fájlról van szó – és olyan betűkészletet használ, amely minden megjelenítendő karakter képét tartalmazza.

2.2.4. Indítófájl párbeszédes üzemmódban

Párbeszédes (interaktív) üzemmódban használva a Pythont, sokszor kívánatos, hogy az értelmező futtatása előtt bizonyos parancsok mindig lefussanak. Ez a PYTHONSTARTUP nevű környezeti változó értékének megadásával történhet; itt kell megadni a futtatni kívánt parancsfájl nevét. Ez a megoldás azonos a UNIX '.profile' fájl megoldásával.

Az indítófájl beolvasása csak párbeszédes üzemmódban történik; nem történik beolvasás ha a parancsok fájlból jönnek, vagy ha bemenetként a '/dev/tty' explicit módon lett megadva (amelyet egyébként párbeszédes üzemmódban az értelmező alaphelyzetben használ). Az indítófájl a futása során ugyanazt a névterületet (name space) használja mint a párbeszédes üzemmód, így a benne definiált vagy bele importált objektumok változatlanul, további pontosítás nélkül használhatók párbeszédes üzemmódban is. Ebben a fájlban a `sys.ps1` és `sys.ps2` értékeinek átírásával változtathatod a promtok értékeit.

Amennyiben az aktuális könyvtárból egy másik indítófájlt is szeretnél futtatni, megteheted a globális indítófájl szerkesztésével, ahogy ezt az alábbi példa mutatja: '`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`'.

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

3. fejezet

Kötetlen bevezető a Pythonba

A következő példákban a kimenetet és a bemenetet az elsődleges és másodlagos készenléti jelek – promptok – (‘`>>> `’ és ‘`... `’) meglétével, illetve hiányával különböztetjük meg. A példák kipróbálásához mindent be kell írnod a promptok után, amikor a prompt megjelenik; azok a sorok, amelyek előtt nincs prompt, a fordító kimenetei.

Jegyezd meg, hogy az önmagában álló másodlagos prompt a példa egyik sorában azt jelenti, hogy nem kell semmit írni a sorba; ez jelzi egy többsoros utasítás végét.

Ennek a kézikönyvnek sok példájában – még azokban is, amelyeket interaktív módon írtunk be – szerepelnek megjegyzések. A Pythonban a megjegyzések kettőskereszttel (hash, ‘#’) kezdődnek és a sor végéig tartanak. Egy megjegyzés lehet sor elején, vagy követhet szóközt, tabulátor-karaktert, de ha egy karakterlánc belsejébe teszed, az nem lesz megjegyzés (lásd a példában!). A kettőskereszt karakter egy karakterláncon belül csak egy kettőskereszt.

Példák:

```
# ez az első megjegyzés
SPAM = 1                 # ez a második megjegyzés
                         # ... és ez a harmadik.
STRING = "# Ez nem megjegyzés."

print STRING   # ennek a kimenete a következő sor:
# Ez nem megjegyzés.
```

3.1. A Python használata számológépként

Próbáljunk ki néhány Python utasítást. Indítsuk el az értelmezőt, és várjuk meg az elsődleges promptot: ‘`>>> `’. (Nem kell sokáig várni.)

3.1.1. Számok

A parancsértelmező úgy működik, mint egy sima számológép: be lehet írni egy kifejezést, és az kiszámolja az értékét. A kifejezések nyelvtana a szokásos: a `+`, `-`, `*` és `/` műveletek ugyanúgy működnek, mint a legtöbb nyelvben (például, Pascal vagy C); zárójeleket használhatunk a csoportosításra. Például:

```
>>> 2+2
4
>>> # Ez egy megjegyzés
... 2+2
4
>>> 2+2  # és egy megjegyzés ugyanabban a sorban, ahol utasítás van
4
>>> (50-5*6)/4
5
>>> # Egészek osztása az eredmény lefelé kerekített értékét adja:
... 7/3
2
>>> 7/-3
-3
```

A C-hez hasonlóan az egyenlőségjellel (‘=’) lehet értéket adni egy változónak. Az értékadás után az értelmező újabb utasításra vár, látszólag nem történik semmi:

```
>>> szélesség = 20
>>> magasság = 5*9
>>> szélesség * magasság
900
```

(Az ékezetek nekem működtek interaktív módban, lehet, hogy a régebbi programváltozatok miatt érdemes kerülni. A a program elején adjuk meg hogy milyen kódtáblával dolgozunk, például windows alatt:

```
>>> # -*- coding: cp1250 -*-
...
```

Linux alatt is működik az ékezetkezelés, a Linux telepítésekor a magyar nyelvet állítottam be, a Pythonban semmit nem kellett állítanom. – A fordító megjegyzése.)

Több változónak egyszerre tudunk értéket adni:

```
>>> x = y = z = 0  # Nulla lesz az x, y és z
>>> x
0
>>> y
0
>>> z
0
```

A programnyelv támogatja a lebegőpontos számábrázolást; azok a műveletek amelyeknél keverednek a típusok, az egészeket lebegőpontossá alakítják:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Complex numbers are also supported; imaginary numbers are written with a suffix of ‘`j`’ or ‘`J`’. Complex numbers with a nonzero real component are written as ‘`(`*real*`+`*imag*`j)`’, or can be created with the ‘`complex(`*real*`,` *imag*`)`’ function.

A Python komplex számokat is tud kezelni – a képzetes részt a ‘`j`’ vagy ‘`J`’ jellel képezhetjük. A komplex számot ‘`(`*valós*`+`*képzetes* `j)`’ alakban vagy ‘`complex(`*valós*`,` *képzetes*`)`’ alakban írhatjuk, ha a képzetes rész nem nulla.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

A komplex számokat gyakran két lebegőpontos számmal ábrázolják – a képzetes és a valós résszel. A *z* komplex számnak ezeket a részeit a *z*`.real` és *z*`.imag` utasításokkal olvashatjuk vissza.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

The conversion functions to floating point and integer (`float()`, `int()` and `long()`) don’t work for complex numbers — there is no one correct way to convert a complex number to a real number. Use `abs(`*z*`)` to get its magnitude (as a float) or `z.real` to get its real part.

A lebegőpontos és egész típusú konverziós függvények (`float()`, `int()` és `long()`) nem működnek komplex számokra. A komplex - valós számok közötti konverziónak több lehetségesmódja is van – ahhoz, hogy egy komplex számból valósat csinálj, használd az `abs(`*z*`)` utasítást, hogy megkapd a nagyságát (lebegőpontosként) vagy a `z.real` utasítást, ha a valós része kell.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can’t convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
>>> float(8) # lebegopontos alakra konvertal. kimenete: 8.0
```

Interaktív módban az utoljára kiírt kifejezés értéke a `'_' (alsóvonás)` változóban van. Így ha a Pythont asztali számológépként használod, akkor egyszerűbb folytatni a számolásokat, például:

```
>>> adó = 12.5 / 100
>>> ár = 100.50
>>> ár * adó
12.5625
>>> ár + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Ezt a változót csak olvasható változóként kezelhetjük. Ne adjunk értéket neki – mert ha adunk, akkor létrehozunk egy független helyi változót azonos névvel, amely meggátolja a beépített változó elérését. (Ha egy globális változó nevével létrehozunk egy helyi változót, akkor az értelmező a helyit használja.

3.1.2. Karakterláncok

A számok mellett a Python karakterláncokkal is tud műveleteket végezni. A karakterláncokat idézőjelek vagy aposztrófok közé kell zárni:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Egy karakerláncot többféle módon bonthatunk szét több sorba. Azt a sort, amelyet folytatni szeretnénk, visszaperjellel (*back-slash*-jellel: \-el) zárjuk le, ezzel jelezve, hogy a következő sor ennek a sornak a logikai folytatása

```
hello = "Ez egy nagyon hosszú karakterlánc, amely\n\
több sorból áll. Ugyanúgy választható el mint C-ben.\n\
    Jegyezd meg, hogy a soreleji szóközök és tabulátorok\
 fontosak.\n"
print hello
```

A soremelést a

```
\n
```

jellel adtuk meg. Ez a következőket fogja kiírni:

```
Ez egy nagyon hosszú karakterlánc, amely
több sorból áll. Ugyanúgy választható el mint C-ben.
    Jegyezd meg, hogy a soreleji szóközök és tabulátorok fontosak.
```

Ha „raw” (nyers, bármiféle konverzió nélkül pontosan azt adja vissza, amit beírsz) karakterláncot szeretnél készítesz, a `\n` karakterpár nem konvertálódik újsor karakterre. A forráskódban lévő összes sortörés és sorvégi \ jel megmarad:

```
hello = r"Ez egy hosszú karakterlánc, amely több sorból áll,\n\
ahogy C-ben csinálnád!"

print hello
```

Ezt fogja kiírni:

```
Ez egy hosszú karakterlánc, amely több sorból áll,\n\
ahogy C-ben csinálnád!.
```

Vagy a karakterláncot hármas idézőjellel vehetjük körül: `"""` vagy `'''`. Ilyenkor az újsort nem kell jelölnöd mert azok jelölés nélkül benne lesznek a karakterláncban.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

Ez a következő kimenetet adja:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Az értelmező ugyanúgy írja ki a karakterlánc-műveletek eredményét, ahogy begépeltük a bemenetre: idézőjelekben, a belső idézőjeleket és más érdekes karaktereket vissza-perjelekkel megvédve, hogy a pontos értéket megmutassa. (Az ékezetes betűknek a kódját adja vissza vissza-perrel az elején. A vissza-perrel kezdődő speciális jelentésű részeket nevezik escape sorozatoknak, escape karaktereknek... – a ford.) A karakterlánc dupla idézőjelben van, ha egyszeres idézőjelet tartalmaz, de kétszereset nem, különben egyszeres idézőjelben. (A később sorrakerülő `print` utasítást használhatjuk a karakterláncok idézőjel és escape sorozatok nélküli kiíratásához.)

Karakterláncokat a `+` művelettel ragaszthatunk össze és `*`-gal ismételhetünk.

```
>>> szo = 'Segít' + 's'
>>> szo
'Segíts'
>>> '<' + szo*5 + '>'
'<SegítsSegítsSegítsSegítsSegíts>'
```

Két egymást követő karakterláncot az értelmező magától összevon; az első sor fentebb lehetne ‘`szo = 'Segít' 's'`’ is; ez csak két a beégetett karakterláncokra (string literal) érvényes, teszőleges karakterlánc-kifejezésekre nem:

```
>>> import string
>>> 'str' 'ing'                   #  <-  helyes.
'string'
>>> string.strip('str') + 'ing'   #  <-  helyes.
'string'
>>> string.strip('str') 'ing'     #  <-  ez nem megy.
  File "<stdin>", line 1
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

A karakterláncokat fel lehet bontani al-karakterláncokra (indexelni); ahogy a C-ben is, a karakterlánc első karaktere a nullás indexű. Nincs külön karakter típus; egy karakter, az egy egységnyi hosszúságú karakterlánc. Ahogy az Icon-ban, az rész-karakterláncokat *szeletelő jelölési mód*dal jelölhetünk ki: a két indexet kettősponttal választjuk el.

```
>>> szo[4]
't'
>>> szo[0:2]
'Se'
>>> szo[2:4]
'gí'
```

A szeletek indexeinek hasznos alapértékei vannak; az elhagyott első index nullát jelent, ha a másodikat hagyjuk el, akkor a karakterlánc végéig tart a szelet.

```
>>> szo[:2]    # Az első két karakter
'Se'
>>> szo[2:]    # Minden karakter, az első kettőt kivéve
'gíts'
```

A C-vel szemben a karakterláncot nem lehet megváltoztatni. Ha egy indexelt helynek értéket adunk, hibaüzenetet kapunk:

```
>>> szo[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> szo[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Jóllehet, új karakterláncot létrehozni a részeik összerakásával könnyen és hatékony módon lehet:

```
>>> 'x' + szo[1:]
'xegíts'
>>> 'Vidít' + szo[5]
'Vidíts'
```

Itt van egy hasznos, az eredeti változóval azonos értékű karakterlánc: `s[:i] + s[i:]` egyenlő az `s`-el.

```
>>> szo[:2] + szo[2:]
'Segíts'
>>> szo[:3] + szo[3:]
'Segíts'
```

A valótlan méretű szeleteket az értelmező okosan kezeli: egy indexet ami túl nagy helyettesíti a karakterlánc méretével, ha az alsó határ nagyobb mint a felső, akkor egy üres karakterlánccal tér vissza.

```
>>> szo[1:100]
'egíts'
>>> szo[10:]
''
>>> szo[2:1]
''
```

Az indexek negatív számok is lehetnek, ekkor jobb oldalról számol. Például:

```
>>> szo[-1]     # Az utolsó karakter
's'
>>> szo[-2]     # Az utolsó előtti
't'
>>> szo[-2:]    # Az utolsó kettő karakter
'ts'
>>> szo[:-2]    # Az összes, kivéve az utolsó kettő
'Segí'
```

Jegyezd meg, hogy a -0 valóban azonos a 0-val, így ez nem jobbról számol!

```
>>> szo[-0]     # (mivel -0 és 0 egyenlőek)
'S'
```

A tartományon kívüli negatív indexeket megcsonkítja a Python, de ez nem működik egyetlen elemmel (azaz egyetlen karakterláncra):

```
>>> szo[-100:]
'Segíts'
>>> szo[-10]    # hiba
Traceback (most recent call last):
  File "<stdin>", line 1
IndexError: string index out of range
```

(Mivel kívül van az index a megengedett tartományon.)

Úgy a legkönnyebb megjegyezni hogy működnek a szeletek, ha azt képzeljük, hogy az indexek a karakterek *közé* mutatnak, az első karakter bal élét számozzuk nullának. Ekkor az *n* karakterből álló karakterlánc utolsó karakterének jobb éle az *n*, például:

```
 +---+---+---+---+---+---+
 | S | e | g | í | t | s |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Az első sorban álló számok adják a $0 \ldots 5$ indexeket a karakterláncban; a második sor mutatja a megfelelő negatív indexeket. Az *i*-től *j*-ig terjedő szelet mindazokat a karaktereket tartalmazza, amelyek az *i* és *j* jelű élek között vannak.

A nem negatív indexek esetén a szelet hossza az indexek különbségével egyenlő, ha mindkettő a valódi szóhatárokon belül van. Például a `szo[1:3]` hossza 2 (`szo[ettől:eddig]`).

Ford.: az indexek valójában egy-egy konkrét karakterre mutatnak, ezért a fenti ábra így helyénvaló:

```
 +---+---+---+---+---+---+
 | S | e | g | í | t | s |
 +---+---+---+---+---+---+
   0   1   2   3   4   5   6
  -6  -5  -4  -3  -2  -1
```

Ebből világosan látszik, hogy a 0. elem az 'S', és a 3. elem az 'í' betű. A Python szeletelési paraméterei pozíciótól pozícióig (`szo[ettől:eddig]`) működik, de az utolsó elem már nem tartozik a halmazba:

```
print szo[0:6]  # a 0. karaktertől a 6.-ig (de 6-os már nem!)
# kiírja a szót.
```

A beépített `len()` függvény a karakterlánc hosszával tér vissza:

```
>>> s = 'legeslegelkáposztásíthatatlanságoskodásaitokért'
>>> len(s)
47
```

See Also:

Sorozat jellegű típusok
(../lib/typesseq.html)
Itt található az Ascii és a Unicode karakterláncok leírása; példák a *sorozat jellegű típusokra*, illetve az ezen típusokkal végezhető műveletek leírása.

Karakterlánc metódusok
(../lib/string-methods.html)
A hagyományos és a Unicode karakterláncok is sok metódussal rendelkeznek az alapvető karakerlánc műveletek, keresés területén.

Karakterlánc formázó műveletek
(../lib/typesseq-strings.html)
A formázó műveletek akkor kerülnek használatba, amikor egy karakterlánc a % operátor baloldali operandusa – további részletek itt olvashatók.

3.1.3. Unicode szövegek

A Python 2.0-tól kezdve egy új adattípust használhatnak a programozók: az Unicode objektumot. Ezt Unicode adatok (lásd http://www.unicode.org) tárolására és feldolgozására használhatjuk. Ez együtt használható a többi karakterláncot tartalmazó objektummal: automatikus átalakítás (konverzió) jön létre, ahol szükséges.

Az Unicode-nak megvan az az előnye, hogy a modern és ősi irat minden karakteréhez egy sorszámot rendel. Korábban csak 256 lehetséges sorszámot használtak a karakterek leírásához, és egy szöveg rendszerint kapcsolódott egy kódlaphoz, amely leképezte a sorszámokat a karakterekre. Ez rengeteg kavarodáshoz vezetett, főleg a programok más nyelvekre fordításakor (internationalization = '`i18n`', azaz '`i`' + 18 karakter + '`n`'). Az Unicode megoldotta a problémát azzal, hogy egyetlen kódlapot definiál a világ valamennyi nyelvéhez.

Unicode karakterláncokat ugyanolyan egyszerű Pythonban létrehozni, mint a hagyományos szövegeket:

```
>>> u'Hello World !'
u'Hello World !'
```

Az idézőjel előtti kis '`u`' mutatja, hogy Unicode karakterláncot kell létrehozni. Ha speciális karaktereket szeretnénk használni a szövegben, ezt megtehetjük Python *Unicode-Escape* kódolásával. A következő példa megmutatja, hogyan:

```
>>> u’Hello\u0020World !’
u’Hello World !’
```

A `\u0020` escape-sorozat jelzi, hogy egy 0x0020 (a szóköz karakter) sorszámú Unicode karaktert kell beszúrni az adott helyen.

Más karaktereket úgy értelmez, hogy az illető karakter sorszámát felhasználja közvetlenül Unicode sorszámként. Ha szabványos Latin-1 kódolású szöveget használunk (ezt a kódolást használják sok nyugat-európai államban) észrevehetjük, hogy az Unicode első 256 karaktere ugyanaz, mint a Latin-1-es kódolás első 256 karaktere.

Szakértőknek: Van egyfajta nyers mód (raw mode), amilyen a normál karakterláncoknál is. Az idézőjel elé ’ur’-et kell írni, hogy a Python a *Raw-Unicode-Escape* kódolást használja. Ez a fenti `\uXXXX` konverziót csak akkor fogja alkalmazni, ha páratlan számú vissza-per jel van a kis ’u’ előtt.

```
>>> ur’Hello\u0020World !’
u’Hello World !’
>>> ur’Hello\\u0020World !’
u’Hello\\\\u0020World !’
```

A nyers mód akkor a leghasznosabb, ha sok vissza-per jelet kell beírni, mint például reguláris kifejezéseknél.

A szabványos kódoláson kívül is számos egyéb mód van a Pythonban Unicode szövegek létrehozására valamely ismert kódolás alapján.

A beépített `unicode()` függvény gondoskodik a használható Unicode kódolók és dekódolók eléréséről. Néhány közismertebb az ilyen kódolások közül, amelyeket átkódolhatunk: *Latin-1*, *ASCII*, *UTF-8*, és *UTF-16*. Az utóbbi kettő változó hosszúságú kódolás, amely minden Unicode karaktert egy vagy két bájton tárol. Az alapértelmezett kódolás az ASCII, amely a 0-tól 127-ig terjedő karaktereket alakítja át, a többit hibautasítással elutasítja. Ha az Unicode karakterláncot kiíratjuk, fájlba írjuk, vagy az `str()` függvénnyel átalakítjuk eszerint a kódolás szerint megy végbe.

```
>>> u"abc"
u’abc’
>>> str(u"abc")
’abc’
>>> u"äöü"
u’\xe4\xf6\xfc’
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: ’ascii’ codec can’t encode characters in position 0-2: ordinal not in ra
```

Ahhoz, hogy a Unicode karakterláncot egy 8-bites karakterlánccá alakítsuk a Unicode objektum egy `encode()` metódussal szolgál, amely egy argumentumot vár, a kódolás nevét. Kisbetűs kódolásneveket használjunk.

```
>>> u"äöü".encode(’utf-8’)
’\xc3\xa4\xc3\xb6\xc3\xbc’
```

Ha van egy speciális kódolású adatunk, és meg szeretnénk kapni a megfelelő Unicode karakterláncot, a `unicode()` függvényt használjuk második argumentumként a kódolás nevével.

```
>>> unicode(’\xc3\xa4\xc3\xb6\xc3\xbc’, ’utf-8’)
u’\xe4\xf6\xfc’
```

3.1.4. Listák

A Python többfajta *összetett* adattípust ismer, amellyel több különböző értéket csoportosíthatunk. A legsokoldalúbb a *lista*, amelyet vesszőkkel elválasztott értékekként írhatunk be szögletes zárójelbe zárva. A lista elemeinek nem kell azonos típusúaknak lenniük.

```
>>> a = [’spam’, ’tojások’, 100, 1234]
>>> a
[’spam’, ’tojások’, 100, 1234]
```

Ahogy a karakterlánc-indexek, úgy a lista-indexek is 0-val kezdődnek, és a listákat is szeletelhetjük, összeilleszthetjük és így tovább:

```
>>> a[0]
’spam’
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
[’tojások’, 100]
>>> a[:2] + [’sonka’, 2*2]
[’spam’, ’tojások’, ’sonka’, 4]
>>> 3*a[:3] + [’Boe!’]
[’spam’, ’tojások’, 100, ’spam’, ’tojások’, 100, ’spam’, ’tojások’, 100, ’Boe!’]
```

A karakterláncokkal ellentétben – amelyek *megváltoztathatatlanok* – a listák egyes elemeit módosíthatjuk:

```
>>> a
[’spam’, ’tojások’, 100, 1234]
>>> a[2] = a[2] + 23
>>> a
[’spam’, ’tojások’, 123, 1234]
```

A szeleteknek értékeket is adhatunk és ez akár a lista elemszámát is megváltoztathatja:

```
>>> # Pár elem átírása:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Pár elem törlése:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Pár elem beszúrása:
... a[1:1] = [’bletch’, ’xyzzy’]
>>> a
[123, ’bletch’, ’xyzzy’, 1234]
>>> a[:0] = a     # Beszúrja magát (pontosabban egy másolatát) a saját elejére.
>>> a
[123, ’bletch’, ’xyzzy’, 1234, 123, ’bletch’, ’xyzzy’, 1234]
```

A beépített `len()` függvény listákra is alkalmazható:

```
>>> len(a)
8
```

A listák egymásba ágyazása is lehetséges:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Nézd meg az 5.1-es szakaszt!
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Figyeld meg, hogy az utolsó példában `p[1]` és `q` valóban ugyanarra az objektumra hivatkozik! Még később visszatérünk az *objektumok értelmezésére*.

3.2. Első lépések a programozás felé

Természetesen a Pythont sokkal összetettebb feladatokra is használhatjuk annál, minthogy kiszámoljuk 2+2 értékét. Például írhatunk egy rövid ciklust a Fibonacci-sorozat kiszámolására:

```
>>> # Fibonacci-sorozat:
... # az előző két elem összege adja a következőt
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Ebben a példában a Python több új tulajdonságát megtaláljuk:

- Az első sor egy *többszörös értékadást* tartalmaz: `a` és `b` egyszerre veszi fel a 0 és 1 értékeket. Az utolsó sorban újból ezt használjuk, hogy megmutassuk, hogy előbb a jobboldal értékelődik ki, és csak azután megy végbe az értékadás. A jobboldali kifejezések jobbról balra értékelődnek ki.
- A `while` ciklus addig hajtódik végre, amíg a feltétel (itt: `b < 10`) igaz marad. A Pythonban – ahogy a C-ben is – minden nullától eltérő egész érték igazat, a nulla hamisat jelent. A feltétel lehet egy karakterlánc vagy egy lista (gyakorlatilag bármilyen sorozat): minden aminek nem nulla a hossza – igaz, az üres sorozatok hamisak. A példában használt feltétel egy egyszerű összehasonlítás. A legalapvetőbb összehasonlító relációkat a C-vel azonosan jelöljük: `<` (kisebb mint), `>` (nagyobb mint), `==` (egyenlőek), `<=` (kisebb vagy egyenlő), `>=` (nagyobb vagy egyenlő) and `!=` (nem egyenlő).
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!)

provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

- A ciklus *magját beljebb húzzuk*: a behúzás a Python jelölése az utasítások csoportosítására. A Pythonnak (még!) nincs intelligens sorszerkesztője, tehát neked kell egy tabulátort vagy (néhány) szóközt beírnod minden behúzott sor elé. Gyakorlatban az összetettebb bemeneteket úgyis szövegszerkesztővel fogod elkészíteni, a legtöbb szövegszerkesztőnek van eszköze az automatikus behúzásra. Ha egy összetett utasítást írunk be párbeszédes (interaktív) módban, azt egy üres sornak kell követnie (mivel az értelmező nem tudja kitalálni, lesz-e még újabb sor). Jegyezd meg, hogy minden sort ugyanannyival kell beljebb húzni.

- A `print` utasítás kiírja annak a kifejezésnek az értékét, amelyet megadtunk. Ez abban különbözik attól, mintha a kifejezést csak önmagában írnánk be (mint ahogy számológépként használtuk), ahogy a többszörös értékeket és a karakterláncokat kezeli. A karakterláncokat idézőjelek nélkül írja ki, és szóközöket illeszt az egyes tagok közé, így széppé teheted a kimenetet:

```
>>> i = 256*256
>>> print 'Az i értéke:', i
Az i értéke: 65536
```

Egy lezáró vessző meggátolja az újsor karaktert a kimeneten:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Vegyük észre, hogy az értelmező újsort illeszt be miellőtt visszaadja a következő promptot, ha nem zárult le a sor.

4. fejezet

További vezérlő utasítások

Az imént említett `while` utasítás mellett a Python ismeri a más nyelvekben szereplő leggyakoribb vezérlő utasításokat is – némi változtatással.

4.1. Az `if` utasítás

Talán a legjobban ismert utasítástípus az `if` utasítás. Példa:

```
>>> x = int(raw_input("Írjon egy számot: "))
>>> if x < 0:
...      x = 0
...      print 'Negatív, lecseréltem nullára'
... elif x == 0:
...      print 'Nulla'
... elif x == 1:
...      print 'Egy'
... else:
...      print 'Egynél több.'
...
```

Hiányozhat, de lehet egy vagy akár egynél több `elif` rész, a `else` rész szintén elmaradhat. Az '`elif`' kulcsszó – amely az 'else if' rövidítése – hasznos a felesleges behúzások elkerülésére. Egy `if` … `elif` … `elif` … sor helyettesíti a más nyelvekben található `switch` és `case` utasításokat.

4.2. A `for` utasítás

A `for` utasítás különbözik attól, amely a C-ben és a Pascalban található. Ahelyett, hogy mindíg egy számtani sorozattal dolgozna (mint a Pascalban), vagy hogy megadná a lehetőséget a felhasználónak, hogy saját maga határozza meg mind az iterációs lépést, mind a kilépési feltételt (ahogy a C-ben van) a Python `for` utasítása végighalad a sorozat (pl. szövegek listája) összes elemén olyan sorrendben, ahogy a listában szerepelnek. Például:

```
>>> # Megmérjük a szavak hosszát:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Nem biztonságos dolog megváltoztatni a sorozatot, amelyen ciklussal végighaladunk (ez csak megváltoztatható sorozattal, például listával történhet meg). Ha mégis szükséges megváltoztatnod a listát, akkor a másolatát használd a for ciklusban. A szelet (slice) jelölési móddal ezt kényelmesen megteheted:

```
>>> for x in a[:]:  # egy másolatot csinál az eredeti listáról
...    if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3. A `range()` függvény

Ha egy számsorozaton kell végighaladnunk, a `range()` beépített függvény lehet szolgálatunkra. Ez egy számtani sorozatot állít elő lista formában, pl.:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A megadott végpont sohasem része a listának; `range(10)` 10 elemű listát hoz létre, pontosan egy tízelemű sorozat indexeit. Lehetőség van rá, hogy a sorozat más számmal kezdődjön, vagy hogy más lépésközt adjunk meg (akár negatívat is):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Ha egy sorozat indexein akarunk végighaladni, használjuk a `range()` és `len()` függvényeket a következőképpen:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

4.4. A `break` és a `continue` utasítások, az `else` ág a ciklusokban

A `break` utasítás – ahogy a C-ben is – a `break`-et tartalmazó legmélyebb `for` vagy `while` ciklusból ugrik ki.

A `continue` utasítás – ez is a C-ből származik – a ciklus további utasításait átugorva a következő elemre ugrik (és elkezdi a következő ciklus-hurkot).

A ciklus-szervező utasításoknak lehet egy `else` águk. Ez akkor hajtódik végre, ha a ciklus végighaladt a listán (`for` esetén), illetve ha a feltétel hamissá vált (`when` esetén), de nem hajtódik végre, ha a ciklust a `break` utasítással szakítottuk meg. Ezt a következő példával szemléltetjük, amely a prímszámokat keresi meg:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'felbontható:', x, '*', n/x
...            break
...     else:
...          print n, 'prímszám.'
...
2 prímszám.
3 prímszám.
4 felbontható: 2 * 2
5 prímszám.
6 felbontható: 2 * 3
7 prímszám.
8 felbontható: 2 * 4
9 felbontható: 3 * 3
```

4.5. A `pass` utasítás

A `pass` utasítás nem csinál semmit. Akkor használható, ha szintaktikailag szükség van egy utasításra, de a programban nem kell semmit sem csinálni. Például:

```
>>> while 1:
...       pass # Elfoglalt - billentyűzetről érkező megszakításra vár.
...
```

4.6. Függvények definiálása

Létrehozhatunk egy függvényt, amely egy megadott értékig írja ki a Fibonacci-sorozatot:

```
>>> def fib(n):    # Kiír egy Fibonacci-sorozatot n-ig
...     "Kiír egy Fibonacci-sorozatot n-ig."
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Hívjuk meg a függvényt amit létrehoztunk:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A `def` kulcsszó a függvény *definícióját* jelzi. Ezt egy függvénynévnek, majd zárójelben a paraméterek listájának

kell követnie. Az utasítások – amelyek a definíció testét alkotják – a következő sorban kezdődnek, és behúzással kell kezdeni azokat. A függvény testének első utasítása lehet egy szöveges 'emberi mondat' is; ez a karakterlánc a függvény dokumentációs karakterlánca, angolul röviden *docstring*.

Vannak eszközök, amelyek a *docstring*-et használják ahhoz, hogy az online vagy a nyomtatott dokumentációt elkészítsék, vagy hogy a felhasználót segítsék a kódban történő interaktív böngészéshez. Bevált gyakorlat, hogy a docstringet beleírjuk a kódba, kérünk téged hogy te is szokjál rá.

A függvény *végrehajtása* egy új szimbólum-táblázatot hoz létre a függvény helyi változói számára. Pontosabban: minden értékadás a függvényben a helyi szimbólum-táblázatban tárolódik; a változókra való hivatkozások esetén először a Python helyi szimbólum-táblázatban, aztán a globális szimbólum-táblázatban, végül a belső (built-in) nevek közt keresgél. Így globális változóknak nem adhatunk közvetlenül értéket egy függvényben (hacsak nem nevezzük meg egy `global` utasításban), jóllehet hivatkozhatunk rá.

Ford.: Ez a kód a fenti leírással ellentétben a 'gyumolcsok' függvényen kívüli változót módosítani tudja. A 'penztarcaban' valtozot olvasni tudjuk, de ha ertekadassal probalkozunk, az olvasast is hibanak jelzi az ertelmezo:

```
gyumolcsok = ["alma"]
penztarcaban = 22


def kosar_kiirasa():
    print gyumolcsok
    gyumolcsok.append('barack')
    print gyumolcsok

    # erdekes: ha csak ez a print sor van, es a try-ban nem adunk
    # erteket, akkor mukodik.
    # ha adunk erteket, hibasnak jeloli ezt a sort is.
    print penztarcaban, "Ft"

    try:
        # penztarcaban = 33
        print penztarcaban, "Ft - try blokkbol, ertekadas kiprobalasnal"
        pass
    except:
        print "a penztarcaban valtozot nem tudtam modositani"

kosar_kiirasa()
print "kosar kiirasa utan: ", gyumolcsok
```

A függvényhívás aktuális paraméterei bekerülnek a hívott függvény helyi szimbólum-táblázatába amikor azt meghívjuk, így az argumentumok mindig *értékeket* adnak át. (ahol az érték mindig az objektumra történő *hivatkozás*, nem az objektum értéke). [1] Ha a függvény egy másik függvényt hív, akkor az új híváshoz egy új helyi szimbólumtábla jön létre.

A függvénydefiníció a függvény nevét beírja az aktuális szimbólumtáblába. A függvénynév értékének van egy típusa, amelyet a fordító a felhasználó által definiált függvényként ismer fel. Ezt az értéket átadhatjuk egy másik változónak, amely ekkor szintén függvényként használható. Ez egy általános átnevezési eljárás:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Kifogásolhatja bárki, hogy a `fib` nem függvény, hanem eljárás. A Pythonban – ahogy a C-ben is – az eljárások

[1] Pontosabban a *objektumra történő hivatkozással történő meghívás* jobb elnevezés lenne erre, mert ha egy megváltoztatható objektumot adunk át, a hívó minden változást látni fog, amit a hívott függvény csinál vele (pl. ha elemet szúr a listába).

olyan függvények, amelyek nem adnak vissza értéket.

Gyakorlatilag egy nagyon különös értéket adnak vissza, ez az érték a `None` – ez egy belső (built-in) név. A `None` érték kiírását általában elnyomja az értelmező, kivéve ha csak ezt az értéket kell kiírnia. Erről meggyőződhetünk, ha akarunk:

```
>>> print fib(0)
None
```

Könnyen írhatunk olyan függvényt, amely visszatér a Fibonacci-sorozat értékeit tartalmazó listával ahelyett, hogy kiíratná azokat.

```
>>> def fib2(n): # Visszaadja a Fibonacci-sorozatot n-ig
...     "A Fibonacci-sorozat n-nél kisebb elemeit adja vissza egy listában."
...     eredmeny = []
...     a, b = 0, 1
...     while b < n:
...         eredmeny.append(b)    # lásd lejjebb
...         a, b = b, a+b
...     return eredmeny
...
>>> f100 = fib2(100)    # hívjuk meg
>>> f100                # írjuk ki az eredményt
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Ez a példa néhány új vonását mutatja a Pythonnak:

- A `return` utasítás egy értékkel tér vissza a függvény futásának bejezésekor. Ha a `return` utasítás paraméter nélkül `None` értéket ad vissza. Ha egy függvény lefutott, de te nem adtál meg `return` utasítást, a függvény `None` értékkel tér vissza.
- A `eredmeny.append(b)` utasítás meghívja az `eredmeny`-lista objektum egy metódusát. A metódus egy olyan függvény, amely egy objektumhoz „tartozik”, `obj.metódusnév` alakban írjuk, ahol az `obj` valamelyik objektum (lehet egy kifejezés), és a `metódusnév` egy olyan metódus neve, amelyet az objektumtípus definiál. Különböző típusoknak különböző metódusai vannak. Különböző típusoknak lehet azonos nevű metódusa mindenféle kétértelműség veszélye nélkül. (Lehetőség van rá, hogy definiáljunk saját objektumokat és metódusokat az *osztályok* használatával, ahogy később azt látni fogjuk az oktatóban.)

  A példában szereplő `append()` metódus a lista-objektumokra lett definiálva; ez hozzáad egy új elemet a lista végéhez. Ebben az esetben azonos az ‘`eredmeny = eredmeny + [b]`’ alakkal, de ez sokkal hatékonyabb.

4.7. Még több tudnivaló a függvények definiálásáról

Lehetőségünk van függvényeket változó argumentummal definiálni. Ennek három formája van, amelyek variálhatók.

4.7.1. Alapértelmezett (default) argumentumértékek

A leghasznosabb alak az, ha egy vagy több argumentumnak is meghatározott alapértéket adunk meg (azaz egy olyan értéket, amit ez az argumentum felvesz, ha nem adunk értéket neki. Ez így egy olyan függvényt hoz létre, amelyet kevesebb argumentummal is meghívhatunk, mint amennyivel definiáltuk:

```
def ask_ok(szoveg, probalkozasok=4, hibauzenet='igen vagy nem!'):
    while 1:
        ok = raw_input(szoveg)
        if ok in ('i', 'igen','I','IGEN'): return 1
        if ok in ('n', 'nem', 'N','NEM'): return 0
        probalkozasok = probalkozasok - 1
        if probalkozasok < 0: raise IOError, 'értelmetlen használat'
        print hibauzenet
```

Ez a függvény ehhez hasonlóan hívható meg: `ask_ok('Valóban ki akarsz lépni?')` vagy így: `ask_ok('Felülírhatom a fájlt?', 2)`.

Az előző program egyben példa az `in` kulcsszó használatára is. Így tesztelhetjük, hogy a sorozat vajon tartalmaz-e egy adott értéket, vagy nem.

Az alapértékeket a fordító akkor határozza meg, amikor a függvény definíciójával először találkozik, emiatt ezek kiszámítása csak egyszer történik meg! *(defining)* Így például a következő program eredménye `5`lesz:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

Fontos figyelmeztetés: Az alapértékeket a fordító **csak egyszer** határozza meg! Emiatt különbség van, ha az alapérték megváltoztatható objektum, mint amilyen a lista, szótár vagy a legtöbb példányosodott osztály. Például az alábbi függvény összegyűjti az egymás utáni hívások során neki adott paramétereket:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

A program kimenete:

```
[1]
[1, 2]
[1, 2, 3]
```

Ha nem akarod az alapértékeket láthatóvá tenni az egymást követő hívások számára, akkor ehhez hasonlóan írd a függvényt:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2. Kulcsszavas argumentumok

A függvényeket akár '*kulcsszó = érték*' formában megadott argumentumok használatával is meghívhatjuk. Például a következő függvény:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "Volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

meghívható az összes alábbi módon:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

de a következő hívások mind érvénytelenek:

```
parrot()                     # a kötelező argumentum hiányzik
parrot(voltage=5.0, 'dead')  # nem-kulcsszavas argumentum kulcsszavas után
parrot(110, voltage=220)     # kétszeres értékadás egy argumentumnak
parrot(actor='John Cleese')  # ismeretlen kulcsszó
```

Általában egy argumentumlistában néhány helyhez kötött argumentum után néhány kulcsszavas argumentumnak kell szerepelnie, ahol a kulcsszavakat a formális paraméterek közül kell választani. Nem lényeges, hogy egy formális paraméternek van-e alapértéke vagy nincs. Egy hívás során nem kaphat egy argumentum egynél több alkalommal értéket – helyhez kötött argumentumhoz tartozó formális paraméter nem használható kulcsszóként ugyanannál a hívásnál. Itt van egy példa, amely nem hajtódik végre emiatt a megkötés miatt:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Ha van egy ***név* alakú formális paraméter utolsóként, akkor egy ilyen nevű szótárban tárolódik az összes kulcsszavas argumentum, amelynek a kulcsszava nem illeszkedik egyetlen formális paraméterre sem. Ez együtt használható egy **név* alakú formális paraméterrel (ez a következő alszakaszban lesz leírva) amely belerakja az összes olyan nem-kulcsszavas argumentumot, amely nincs benne a formális paraméterlistában, egy tuplébe. (A **név*-nek mindíg a ***név* előtt kell lennie.) Például, ha egy ilyen függvényt definiálunk:

```
def sajtuzlet(sajtfajta, *argumentumok, **kulcsszavak):
    print "-- Van Önöknél néhány", sajtfajta, '?'
    print "-- Sajnálom, teljesen kifogytunk a", sajtfajta+'ból'
    for arg in argumentumok: print arg
    print '-'*40
    for kw in kulcsszavak.keys(): print kw, ':', kulcsszavak[kw]
```

Ez meghívható így is:

```
sajtuzlet('Pálpusztai', "Ez nagyon büdös, uram.",
          "Ez nagyon, NAGYON büdös, uram.",
          vevo='Sajti János',
          boltos='Pálinkás Mihály',
          helyszin='Sajtbolt')
```

és természetesen ezt fogja kiírni:

```
-- Van Önöknél néhány Pálpusztai ?
-- Sajnálom, teljesen kifogytunk a Pálpusztaiból
Ez nagyon büdös, uram.
Ez nagyon, NAGYON büdös, uram.
----------------------------------------
vevo : Sajti János
boltos : Pálinkás Mihály
helyszin : Sajtbolt
```

Megjegyzendő, hogy a `kulcsszavak` nevű szótár tartalmának kinyomtatása előtt hívtuk meg a kulcsszó-argumentum neveinek listájához tartozó `sort()` eljárást; ha nem ezt tesszük, akkor az argumentumok listázási sorrendje határozatlan.

4.7.3. Tetszőleges hosszúságú argumentumlisták

Végül itt a legritkábban használt lehetőség, amikor egy függvénynek tetszőleges számú argumentuma lehet. Ezeket az argumentumokat egy tuplébe helyezi el a Python. A változó számosságú argumentum előtt akárhány (akár egy sem) egyszerű argumentum is előfordulhat.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4. Argumentumlista kicsomagolása

Ennek fordítottja történik, ha listába vagy tuplébe becsomagolt argumentumokat ki kellene csomagolni olyan függvény meghívásához, amely elkülönített, helyhez-kötött változókat vár. Például a beépített `range()` függvény egymástól elkülönítve várja a *start* és *stop* értékeket. Ha ezek nem egymástól elválasztva állnak rendelkezésre, akkor a függvényhívásban a `*` műveletjelet tegyük az összetett-típusú változó neve elé, ez kicsomagolja a listából vagy tupléből az adatokat.

```
>>> range(3, 6)      # normális függvényhívás, különálló paraméterekkel
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)     # listából kicsomagolt paraméterekkel történő függvényhívás
[3, 4, 5]
```

4.7.5. Lambda formák

Többek kívánságára néhány olyan vonás került a Pythonba, amelyek a funkcionális programozási nyelvekben és a Lisp-ben is megtalálhatóak. A `lambda` kulcsszóval rövid névtelen függvényeket lehet létrehozni. Íme egy függvény, amely a két argumentumának összegével tér vissza: '`lambda a, b: a+b`'. A lambda formákat

mindenhol használhatjuk, ahol függvény objektumok szerepelhetnek.

Szintaktikailag egyetlen kifejezés szerepelhet bennük. Általánosságban tekintve 'hab' a normális függvények 'tortáján'. Beágyazott függvénydefinícióként látja az őt meghívó környezet minden változóját.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42) # make_incrementor magyarul kb.: csinálj növelő-t
>>> f(0)
42
>>> f(1)
43
```

4.7.6. A dokumentációs szövegek

A dokumentációs szövegek tartalmával és formájával kapcsolatban egy kialakult és bevált szokásról beszélhetünk.

Az első sor mindig az objektum céljának rövid, tömör összegzése. Rövidsége miatt nem kell tartalmaznia az objektum nevét vagy típusát, hiszen ezek az adatok más úton is kinyerhetők (kivéve, ha az objektum neve a függvény működését leíró ige). A szöveg nagybetűvel kezdődik és ponttal végződik.

Ha a dokumentációs szöveg (`docstring`) több sorból áll, a második sor üres lesz – ezzel vizuálisan elkülönítjük a fejrészt/összegzést a leírás további részétől. Az üres sort egy vagy több rész követheti, ahol leírjuk az objektum hívásának módját, a mellékhatásokat stb.

Maga a Python értelmező nem szedi le a helyközöket a többsoros beégetett szövegből – ha ezek kiszűrése szükséges, akkor ehhez külön szövegfeldolgozó progit kellene használni. Ezt a problémát a következő konvenció használatával kezeljük. Az első sor *után* a legelső nem üres sorban megjelenő szöveg behúzási távolsága határozza meg az egész dokumentációs szöveg behúzását. (A legelső sort azért nem hassználjuk erre a célra, mert a szöveg első betűje általában szorosan követi a karakterláncot nyitó macskakörmöt, ennek eltolása nem lenne nyilvánvaló dolog.) A `docstring` – fejrészt követő minden első sorának elejéről levágunk pont ennyi helyközt. Ha ennél kevesebb helyközt tartalmaz valamely sor – bár ilyennek nem kéne lennie – csak a helyközök törlődnek, karakter nem vész el. A behúzások egyenlőségét ajánlott mindig a tabulátorokat kibontva ellenőrizni (általában 1 tabulátort 8 helyközzel helyettesítünk).

Itt van egy példa a többsoros docstring-re:

```
>>> def fuggvenyem():
...     """Nem csinál semmit, de ez dokumentálva van.
...
...     Valóban nem csinál semmit.
...     """
...     pass
...
>>> print fuggvenyem.__doc__
Nem csinál semmit, de ez dokumentálva van.

     Valóban nem csinál semmit.
```

5. fejezet

Adatstruktúrák

Ez a fejezet az eddig tanultakból pár dolgot részletesebben is leír, és pár új dolgot is megmutat.

5.1. Még több dolog a listákról

A lista adattípusnak a már megismerteken kívül több eljárása (method) is van. Az összes eljárás ismertetése:

append(*x*)
Egy elemet hozzáad a lista végéhez; megegyezik az `a[len(a):] = [x]` utasítással.

extend(*L*)
A lista végéhez hozzáfűzi az L listát (mindegyik elemét egyenként); ugyanaz, mint az `a[len(a):] = L`.

insert(*i, x*)
Beszúr egy elemet az adott helyre. Az első argumentum az elem indexe, amely elé beszúrjuk, így `a.insert(0, x)` a lista elejére szúr be, és az `a.insert(len(a), x)` ugyanazt jelenti mint az `a.append(x)`.

remove(*x*)
Eltávolítja a legelső olyan elemet a listából, amelynek értéke `x`. Hiba, ha nincs ilyen.

pop([*i*])
Eltávolítja az adott helyen lévő elemet a listából, és visszaadja az értékét. Ha nem adtunk meg indexet, akkor az `a.pop()` az utolsó elemmel tér vissza. (Ekkor is eltávolítja az elemet.) (A függvény-argumentum megadásánál használt szögletes zárójel azt jelenti, hogy a paraméter megadása tetszőleges, és nem azt, hogy a [] jeleket be kell gépelni az adott helyen. Ezzel a jelöléssel gyakran találkozhatsz a *Python Standard Library-ban (Szabványos Python könyvtárban)*)

index(*x*)
Visszatér az első olyan elem indexével, aminek az értéke `x`. Hiba, ha nincs ilyen.

count(*x*)
Visszaadja `x` előfordulásának a számát a listában.

sort()
Rendezi a lista elemeit. A rendezett lista az eredeti listába kerül.

reverse()
Megfordítja az elemek sorrendjét a listában - szintén az eredeti lista módosul.

Egy példa, amely tartalmazza a legtöbb eljárást:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

5.1.1. Lista használata veremként

A lista eljárásai megkönnyítik a lista veremként (stack) történő használatát ahol az utolsó lerakott elemet vesszük ki először („utoljára be, először ki”, LIFO). Ahhoz, hogy a verem tetejére egy elemet adjunk, használjuk az `append()` utasítást.

A lista legelső/legfelső elemének kivételéhez használjuk a `pop()` utasítást mindenféle index nélkül. Például:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. A Listák használata sorként (queue)

A listát használhatjuk úgy is mint egy sort, ahol az első hozzáadott elemet vesszük ki először („first-in, first-out”, FIFO). Ahhoz, hogy elemet hozzáadjunk a sor végéhez, használjuk az `append()` utasítást. A sor első elemét visszakaphatjuk a `pop()` utasítással, ha az argumentuma `0`. Például:

```
>>> sor = ["Maca", "János", "Mihály"]
>>> sor.append("Teri")           # Teri megérkezett
>>> sor.append("Gergely")          # Gergely megérkezett
>>> sor.pop(0)
'Maca'
>>> sor.pop(0)
'János'
>>> sor
['Mihály', 'Teri', 'Gergely']
```

5.1.3. Eszközök a funkcionális programozáshoz

Három olyan beépített függvényünk van, amelyek nagyon hasznosak a listáknál – ezek a `filter()`, `map()`, és a `reduce()`.

A '`filter(`*függvény*`, `*sorozat*`)`' egy – lehetőség szerint azonos típusú – sorozattal tér vissza, mely a sorozatnak azokat az elemeit tartalmazza, amelyekre *függvény*`(`*elem*`)` igaz. Például a következő módon szűrhetjük ki a 3-mal és 2-vel nem osztható számokat:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

A '`map(`*függvény*`, `*sorozat*`)`' kiszámolja a *függvény*`(`*elem*`)` értéket a sorozat minden elemére és az eredmények listájával tér vissza. Például így számolhatjuk ki az első néhány köbszámot:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Egynél több sorozatot is feldolgozhatunk; a függvénynek ekkor annyi argumentumának kell lennie, ahány sorozat van. Ekkor a függvényt az egymáshoz tartozó értékpárokkal hívja meg a Python (vagy `None`-t ad, ha valamelyik sorozat rövidebb a másiknál). Ha a függvény helyére `None`-t írunk, akkor a függvény a saját argumentumait adja vissza.

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

A két esetet összevetve láthatjuk, hogy a '`map(None, `*lista1*`, `*lista2*`)`' elegáns módja annak, hogy két listát párok listájává alakítsunk. Például:

```
>>> sor = range(8)
>>> def negyzet(x): return x*x
...
>>> map(None, seq, map(negyzet, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

A '`reduce(`*függv*`, `*sorozat*`)`' egyetlen értékkel tér vissza, melyet úgy kapunk, hogy a bináris, kétváltozós *függv*

függvényt meghívjuk a sorozat első két elemével, majd az eredménnyel és a következő elemmel, és így tovább. Például az 1-től 10-ig terjedő egész számokat így adhatjuk össze:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Ha csak egy érték van a sorozatban, akkor azt az értéket kapjuk; ha a sorozat üres, kivételdobás történik.

A harmadik argumentum használatával megadhatjuk a kezdőértéket. Ekkor úgy működik, mintha a sorozat legelején ez az érték állna. Ekkor tehát egy üres sorozat a kezdeti értéket adja vissza. Például:

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Ne használd a példában definiált `sum()` függvényt: mivel számok összegzésére gyakran van igény, a beépített `sum(`*sequence*`)` függvényt használjuk erre, amely pontosan ugyanúgy működik, mint a fenti példában definiált függvény. New in version 2.3.

5.1.4. A listák magas szintű használata

Ha megértjük, hogyan működnek a listák, akkor rövid és átlátható programokat készíthetünk listafeldolgozásra a `map()`, `filter()` vagy `lambda` függvények használata és az ezt követő újrarendezés nélkül is. Az eredményként kapott listadefiníció gyakran világosabb, mintha a beépített eljárásokkal kaptuk volna. Minden magas fokú listakezelés így épül fel: a kifejezést egy `for` ág követi, ezt pedig nulla vagy több `for` vagy `if` ág. Az eredmény egy olyan lista, amely a `for` és `if` ágak alapján a kifejezés kiértékelésével keletkezik. Ha a kifejezés tuple típust eredményez, akkor zárójelbe kell tenni a kifejezést.

```
>>> freshfruit = [’  banana’, ’  loganberry ’, ’passion fruit  ’]
>>> [weapon.strip() for weapon in freshfruit]
[’banana’, ’loganberry’, ’passion fruit’]
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec] # hiba - a tuple típus zárójelet kíván
  File "<stdin>", line 1
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

A magasfokú listakezelés a `map()` eljárásnál sokkal rugalmasabb, és egynél több argumentummal rendelkező függvényekre, valamint beágyazott (nested) függvényekre is alkalmazható:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
[’3.1’, ’3.14’, ’3.142’, ’3.1416’, ’3.14159’]
```

5.2. A `del` utasítás

Egy listaelem eltávolításának egyik módja, hogy az elem értéke helyett az indexét adjuk meg: ez a `del` utasítás. Ez arra is használható, hogy szeleteket töröljünk a listából (amit már megtettünk ezelőtt úgy, hogy a szeletnek az üres lista értékét adtuk). Például:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
```

A `del` utasítást arra is használhatjuk, hogy az egész változót töröljük:

```
>>> del a
```

A továbbiakban hibát generál, ha az `a` névre hivatkozunk (kivéve, ha új értéket adunk neki). Más alkalmazásával

is találkozunk később a `del` utasításnak.

5.3. Tuplék és sorozatok

Láttuk, hogy a listáknak és a karakterláncoknak rengeteg közös tulajdonsága van, például az indexelés és a szeletek használata. Ez két Mindkettő példa a *sorozat*../lib/typesseq.html-adattípusra. Mivel a Python egy folyamatos fejlődésben lévő nyelv, másfajta sorozat adattípusok is hozzáadhatóak. Egy másik beépített sorozat-adattípusa a tuple[1].

A tuple objektumokat tartalmaz vesszőkkel elválasztva, például:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # A tuplékat egymásba ágyazhatjuk:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Az objektumok különböző típusúak is lehetnek. A tuple nem megváltoztatható adattípus, viszont lehetnek megváltoztatható elemei. Példa:

```
>>> megtanulni = ['matek', 'angol'] # Ez egy megváltoztatható lista, [] jelölés!
>>> orarendem = ('tesi', 'nyelvtan', megtanulni) # ket string elem, es egy lista a tuplében

>>> orarendem[0] = 'rajz' # az első órát át akarom írni, nem lehet
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> orarendem[2][0]='versek' # viszont a tuple lista elemén belül -- ami megváltoztatható tí
```

Ahogy látható, a kimeneten a tuplék mindig zárójelezve vannak, így azok egymásba ágyazva is helyesen értelmezhetők; megadhatjuk zárójelekkel és anélkül is, néha azonban feltétlenül szükségesek a zárójelek (amikor az egy nagyobb kifejezés része).

A tuple típust sokféle dologra felhasználhatod. Például: (x, y) koordinátapár tárolása, dolgozók rekordjai egy adatbázisban... A tuplék a karakterláncokhoz hasonlóan megváltoztathatatlanok: nem adhatunk értéket egyetlen elemének (további hasonlóságok vannak a szeleteléssel és az összefűzéssel kapcsolatban is). Létrehozható olyan tuple, amely megváltoztatható elemeket – például tömböket – tartalmaz.

Egy különös probléma nulla vagy egy elemet tartalmazó tuple létrehozása: a nyelv szintaxisa lehetővé teszi ezt. Az üres zárojellel hozható létre a nulla elemű; az egy elemű pedig az érték után tett vesszővel (nem elég, ha az értéket zárójelbe tesszük). Csúnya, de hatékony. Például:

[1] A lista és a tuple hasonló a PHP tömb típusához – a lista egy írható tömb, a tuple egy csak olvasható tömbnek felel meg első közelítésre.

```
>>> ures = ()
>>> egyszeres = 'hello',    # <-- figyeljünk a vesszőre a végén
>>> len(ures)
0
>>> len(egyszeres)
1
>>> egyszeres
('hello',)
```

A `t = 12345, 54321, 'hello!'` értékadás egy példa a tuple változó feltöltésére[2] és újbóli a `12345`, `54321` és `'hello!'` különböző objektumok értékei egy tuplába kerülnek. A fordított művelet is lehetséges, például:

```
>>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking*. Sequence unpacking requires that the list of variables on the left have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking!

Ezt hívják, elég helyesen, *sorozat szétbontásnak.* A sorozat szétbontásához az szükséges, hogy a bal oldalon annyi elem szerepeljen, ahány elem van a tupléban. Jegyezzük meg, hogy a többszörös értékadás csak egy változata a tuple lehetséges feltöltésének!

Van egy kis asszimetria itt: a különböző objektumokkal való feltöltés mindig tuplét eredményez, ugyanakkor a szétbontás minden sorozatra működik.

5.4. A halmazok (Set)

A Python a set adattípust (sets) is alaptípusként definiálja. a set: elemek rendezetlen halmaza, amelyben minden elem csak egyszer fordulhat elő. Alapvető használata: megadott elem meglétének ellenőrzése, elemek kettőzésének kiszűrése. A set objektumok támogatják az olyan matematikai műveleteket, mint az egyesítés (union), közös metszet (intersection), különbség (difference), és a szimmetrikus eltérés (symmetric difference).

Íme egy rövid bemutató:

[2] Ha a változó már tartalmazott objektumokat, az összes eleme törlődik, és az új – feltöltésnél megadott elemeket tartalmazza csak

```
>>> kosar = ['alma', 'narancs', 'alma', 'korte', 'narancs', 'banan']
>>> gyumolcsok = set(kosar)           # set létrehozása egyedi elemekkel
>>> gyumolcsok
set(['narancs', 'korte', 'alma', 'banan'])
>>> 'narancs' in gyumolcsok           # gyors ellenőrzés: benne van-e
True
>>> 'kakukkfu' in gyumolcsok
False

>>> # Példa: set műveletek két szó egyedi betűin
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # 'a'-nak egyedi elemei
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # 'a'-ban megvan, b-ből hiányzik
set(['r', 'd', 'b'])
>>> a | b                             # vagy 'a'-ban, vagy 'b'-ben megvan
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # 'a'-ban és 'b'-ben is megvan
set(['a', 'c'])
>>> a ^ b                             # vagy 'a'-ban, vagy 'b'-ben megvan, de
       # egyszerre mindkettőben nem
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.5. Szótárak

Egy másik hasznos adattípus a Pythonban a *szótár*. A szótárakat más nyelvekben „asszociatív tömböknek” nevezik. Szemben a sorozatokkal – amelyek számokkal vannak indexelve – a tömböket *kulcsokkal* indexeljük, amely mindenféle megváltoztathatatlan típus lehet; karakterláncok és számok mindig lehetnek kulcsok. Tuplék is használhatók kulcsnak, ha csak számokat, karakterláncokat vagy tuplékat tartalmaznak; ha egy tuple megváltoztatható objektumot tartalmaz – közvetlenül vagy közvetve, akkor nem lehet kulcs.

Listát nem lehet kulcsként használni, mert annak értékei az `append()`, `extend()` eljárásokkal, valamint a szeletelő vagy indexelt értékadásokkal (helyben) módosíthatók.

Gondoljunk úgy a szótárra, mint *kulcs: érték* párok rendezetlen halmazára, azzal a megkötéssel, hogy a szótárban a kulcsoknak egyedieknek kell lenniük. Egy kapcsos zárójelpárral egy üres szótárat hozhatunk létre: `{}`. Ha a zárójelbe vesszőkkel elválasztott kulcs:érték párokból álló listát helyezünk, akkor ez belekerül a szótárba; egy szótár tartalma is ilyen módon jelenik meg a kimeneten.

A legfontosabb műveletek egy szótáron: eltárolni egy értéket egy kulccsal együtt, visszakapni egy értéket megadva a kulcsát. Lehet törölni is egy kulcs:érték párt a `del`-lel. Ha olyan kulccsal tárolsz egy új értéket, amilyen kulcsot már használtál, a kulcs az új értékre fog vonatkozni, a régi érték elveszik. Hiba, ha egy nemlétező kulcsra hivatkozol.

A szótár objektum `keys()` eljárása a kulcsok listáját adja vissza véletlenszerű sorrendben (ha rendezni akarod, használd a `sort()` eljárást a kulcsok listájára). Ha ellenőrizni szeretnéd, vajon egy kulcs benne van-e a szótárban, használd a szótárak `has_key()` eljárását.

Íme egy kis példa a szótár használatára:

```
>>> tel = {'János': 4098, 'Simon': 4139}
>>> tel['Géza'] = 4127
>>> tel
{'Simon': 4139, 'Géza': 4127, 'János': 4098}
>>> tel['János']
4098
>>> del tel['Simon']
>>> tel['Pisti'] = 4127
>>> tel
{'Géza': 4127, 'Pisti': 4127, 'János': 4098}
>>> tel.keys()
['Géza', 'Pisti', 'János']
>>> tel.has_key('Géza')
True
```

A `dict()` konstruktor közvetlenül tuplékban tárolt kulcs-érték párok listájából is létre tudja hozni a szótárat. Ha a párok valamilyen mintát követnek, akkor lista-műveletekkel rövidebb módon is megadhatjuk a kulcs-érték listát.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Later in the tutorial, we will learn about Generator Expressions which are even better suited for the task of supplying key-values pairs to the `dict()` constructor.

Az oktató egy későbbi részben tanulni fogunk a 'Generátor kifejezésekről' (Generator Expressions), melyek még alkalmasabbak a kulcs-érték párok beillesztésére, mint a `dict()` konstruktor.

5.6. Ciklustechnikák

Ha végig szeretnénk menni egy szótár elemein, akkor az `iteritems()` eljáással lépésenként egyidőben megkapjuk a kulcsot, és a hozzá tartozó értéket.

```
>>> lovagok = {'Gallahad': 'a tiszta', 'Robin': 'a bátor'}
>>> for k, v in lovagok.iteritems():
...     print k, v
...
Gallahad a tiszta
Robin a bátor
```

Ha sorozaton megyünk végig, akkor az pozíciót jelző index értékét és a hozzá tartozó értéket egyszerre kaphatjuk meg az `enumerate()` eljárással.

Példa:[3]

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

[3] A tic-tac-toe az amőba játék angol nyelvterületen elterjedt neve.

Két vagy több sorozat egyszerre történő feldolgozásához a sorozatokat a `zip()` függvénnyel kell párba állítani.

```
>>> kerdesek = ['neved', 'csoda, amit keresel', 'kedvenc szined']
>>> answers = ['Lancelot', 'A szent Gral', 'Kek']
>>> for q, a in zip(kerdesek, valaszok):
...     print 'Mi a %s?  %s.' % (q, a)
...
Mi a neved?  It is lancelot. Lancelot.
Mi a csoda, amit keresel?  A szent Gral.
Mi a kedvenc szined?  Kek.
```

Egy sorozaton visszafelé haladáshoz először add meg a sorozatot, majd utána hívd meg a `reversed()` függvényt.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Rendezett sorrendben való listázáshoz használd a `sorted()` függvényt, amely új, rendezett listát ad vissza, változatlanul hagyva a régi listát.

```
>>> kosar = ['alma', 'narancs', 'alma', 'korte', 'narancs', 'banan']
>>> for f in sorted(set(basket)):
...     print f
...
alma
banan
korte
narancs
```

5.7. Még több dolog a feltételekről

A `while` és az `if` utasításokban eddig használt feltételek egyéb műveleteket is tartalmazhatnak az összehasonlítás mellett.

Az `in` és `not in` relációk ellenőrzik, hogy az érték előfordul-e egy sorozatban. Az `is` és `is not` relációk összehasonlítják, hogy két dolog valóban azonos-e; ez csak olyan változékony dolgoknál fontos, amilyenek például a listák. Minden összehasonlító relációnak azonos precedenciája van, mely magasabb mint a számokkal végzett műveleteké.

Relációkat láncolhatunk is, például: az `a < b == c` megvizsgálja, hogy az `a` kisebb-e mint `b`, és ezen felül `b` egyenlő-e `c`-vel.

A relációkat összefűzhetjük `and` és `or` logikai műveletekkel is, és a reláció erdményét (vagy bármely logikai műveletét) ellentettjére változtathatjuk a `not` művelettel. Ezeknek mindnek kisebb precedenciájuk van, mint a relációknak, és közülük a `not` rendelkezik a legmagasabbal és az `or` a legkisebbel. Tehát az `A and not B or C` ugyanazt jelenti, mint az `(A and (not B)) or C`. Természetesen a zárójeleket használhatjuk a kívánt feltétel eléréséhez.

Az `and` és `or` logikai műveletek úgynevezett shortcat (lusta/rövid kiértékelésű) műveletek: Az argumentumaik balról jobbra fejtődnek ki, és a kifejtés rögtön megáll, mihelyt a végeredmény egyértelmű. Például: ha `A` és `C`

mindkettő igaz de `B` hamis, akkor a `A és B és C` kifejezés során a C értékét a Python már nem vizsgálja. Általában a shortcut műveletek visszatérési értéke – ha általános értékeket és nem logikai értéket használunk – az utolsónak kifejtett argumantummal egyezik.

Lehetséges, hogy egy reláció vagy más logikai kifejezés értékét egy változóba rakjuk. Például:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Jegyezzük meg, hogy a Pythonban – szemben a C-vel – nem lehet értékadás egy kifejezés belsejében. Lehet, hogy a C programozók morognak emiatt, de így kikerülhető egy gyakori probléma, amelyet a C programokban gyakran elkövetnek: = jelet írnak ott, ahol == kellene.

5.8. Sorozatok és más típusok összehasonlítása

Sorozat objektumokat összehasonlíthatunk másik azonos típusú objektummal. Az összehasonlítás a *lexikografikai* rendezést használja: először az első elemeket hasonlítja össze, ha ezek különböznek, ez meghatározza az összehasonlítás eredményét; ha egyenlőek, akkor a második elemeket hasonlítja össze, és így tovább, amíg az egyik sorozatnak vége nem lesz. Ha a két összehasonlítandó elem azonos típusú sorozat, akkor az összehasonítás rekurzívan történik. Ha a két sorozat minden eleme azonos, akkor tekinthetőek a sorozatok egyenlőeknek. Ha sz egyik sorozat a másiknak kezdő rész-sorozata, akkor a rövidebb sorozat a kisebb. A karakterláncok lexikografikai elemzésére az egyes karakterek ASCII rendezését alkalmazzuk. Néhány példa azonos típusú sorozatok összehasonlítására:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Jegyezzük meg, hogy különböző típusú objektumok összehasonlítása is szabályos. A kimenet jól meghatározott, de önkényes: a típusok a típusneveik szerint rendeződnek. Így egy lista mindig kisebb, mint egy karakterlánc, egy karakterlánc mindig kisebb, mint egy tuple, stb. [4] Vegyes számtípusok a számértékeik szerint rendeződnek, így például 0 egyenlő 0.0-val.

[4] A különböző típusú objektumok összehasonlítási szabályai nem megbízhatóak – a Python későbbi verzióiban változhatnak.

6. fejezet

Modulok (2.4 doc)

Ha a Pythont interaktív módban használod, és kilépsz az értelmezőből, majd újra visszalépsz, minden függvénydefiníció és definiált változó elvész. Emiatt ha nagyobb programot akarsz írni, jobban jársz ha valamilyen szerkesztő programot használsz az értelmező számára előkészteni a bemeneti adatokat, és az értelmezőt úgy futtatod, hogy a bemenet a szövegfájlod legyen.

Ezt a folyamatot *script* írásnak nevezik. Ahogy a programod egyre hosszabb lesz, előbb-utóbb részekre (fájlokra) akarod majd bontani, a könnyebb kezelhetőség végett. Valószínűleg lesznek praktikus függvényeid, amit már megírt programjaidból szeretnél használni a függvénydefiníciók másolása nélkül.

Ennek a támogatására a Python el tudja helyezni a függvénydefiníciókat egy adott fájlba, amik aztán elérhetők lesznek egy szkriptből, vagy az interaktív értelmezőből. Ezeket a fájlokat *modul*oknak hívjuk. A modulban használt definíciók *importálhatók* más modulokba (például a modulok hierarchiájában legfelül lévő *main* modulba is). (A következő példákhoz: a felhasznált változók mind a legfelső névtérben helyezkednek el, a modulok függvényeit itt futtatjuk, interaktív módban.)

A modul egy olyan szöveges file, ami Python definíciókat és utasításokat tartalmaz. A file neve egyben a modul neve is (a '.py' kiterjesztést nem beleértve). A programból az aktuális modul neve a `__name__` globális változóból elérhető (karakterláncként).

Kérlek egy neked megfelelő szövegszerkesztővel hozd létre a 'fibo.py' fájlt, a következő tartalommal:

```
# Fibonacci szamok modul

def fib(n):    # kiirja a Fibonacci sorozatot n-ig
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # visszater a Fibonacci sorozattal, n-ertekig
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Most lépj be a Python értelmezőbe, és importáld a fenti modult a következő paranccsal:

```
>>> import fibo
```

Ez a parancs a `fibo`-ban definiált függvényneveket közvetlenül nem emeli be az aktuális szimbólumtáblába; csak magát a modul nevét, `fibo`-t emeli be. (ford.: mivel a függvények a `fibo` modul részei, ezért elérhetővé válnak

fibo-n keresztül: fibo.fib(szam))

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Ha egy függvényt gyakran szeretnél használni az importált modulból, hozzárendelheted azt egy lokális függvénynévhez:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. A modulokról bővebben...

A modulok végrehajtható utasításokat ugyanúgy tartalmazhatnak, mint függvénydefiníciókat.

Ezeket az utasításokat rendszerint a modul inicializálásához használjuk. Kizárólag a modul *első* importálásakor futnak le. [1]

Minden modulnak megvan a saját szimbólumtáblája, ami a modulban definiált függvények számára globális. Emiatt a modul létrehozója nyugodtan használhatja a modulban lévő globális változókat, és nem kell aggódnia egy esetleges névütközés miatt. (hiába ugyanaz a neve két változónak, ha külön névtérben vannak, nem írják felül egymás értékét)

Másrészről ha pontosan tudod hogy mit csinálsz, el tudod érni a modul globális változóit a változónév teljes elérési útjának a használatával: `modulnev.elemnev`.

A modulok importálhatnak más modulokat. Szokás – de nem kötelező – az `import` utasításokat a modul elején elhelyezni (vagy a szkript elején, lásd később). Az importált modulneveket az értelmező az importáló modul globális szimbólumtáblájába helyezi el.

Az `import` utasítás egyik felhasználási módja, hogy az importált függvényeket közvetlenül az importáló modul szimbólumtáblájába helyezzük el. Például:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Az előbbi példa a helyzi szimbólumtáblába nem helyezi el a modul nevét, amiből az importálás történt. (`fibo` objektum nem jön létre a névtérben)

Ha a modulban lévő összes nevet közvetlenül a helyi szimbólumtáblába szeretnéd importálni, így tudod megtenni:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ha így használod az `import` utasítást, a forrásmodul minden nevét a helyi szimbólumtáblába emeled, kivéve az aláhúzással kezdődőeket (_).

[1] Valójában a függvénydefiníciók szintén végrehajtható utasítások; A függvény neve a végrehajtás során kerül bele a modul globális szimbólumtáblájába.

6.1.1. A Modulok keresési útvonala

Amikor a `spam` modult importáljuk, az értelmező először az aktuális könyvtárban keresi a ‘spam.py’ fájlt. Ha itt nem találja, tovább keres a PYTHONPATH környezeti változóban felsorolt könyvtárakban. Ennek a változónak a szintaktisa ugyanolyan, mint a PATH héj-változónak (Linuxon). Egyszerűen könyvtárnevek listáját tartalmazza. Ha a PYTHONPATH változó nincs beállítva, vagy az abban felsorolt könyvtárak a keresett fájlt nem tartalmazzák, a keresés az alapértelmezett könyvtárban folytatódik. Ez már függ az installálástól, UNIX-on rendszerint ‘.:/usr/local/lib/python’.

Jelenleg a modulokat az értelmező a `sys.path` változóban felsorolt könyvtárakban keresi. Ez a változó az aktuális könyvtárból (ahonnan a programot futtatjuk), a PYTHONPATH-ban lévő könyvtárakból, és az installálástól függő alapértelmezett könyvtárból áll. Így a Python programoknak módjuk nyílik a modulok keresési útvonalát módosítani, vagy akár teljesen kicserélni.

Fontos megemlíteni, hogy mivel az aktuális könyvtár is a keresési útvonal része, a programnak még véletlenül se legyen ugyanaz a neve, mint valamely standard modulnak. Ebben az esetben ugyanis a Python megkísérli a programot modulként betölteni annak importálásakor, ami hibaüzenethez vezet. További információk: 6.2, „Standard Modules,”.

6.1.2. „Lefordított” Python állományok

Fontos a standard modulokat használó programok indulási idejét lerövidíteni. Ha ‘spam.pyc’ fájl létezik abban a könyvtárban, ahol ‘spam.py’ is megtalálható, az értelmező feltételezi, hogy a ‘.pyc’ fájl a ‘.py’ fájlnak az előre lefordított változata. (A Python a .py fájlokat először .pyc bájtkódra fordítja, ami már egy bináris, ember számára nem értelmes, olvasható állomány – ezt hajtja végre az értelmező. Ez a működés a Java nyelvhez hasonló konstrukció)

A fordítás során a ‘.pyc’ fájl ugyanazzal a módosítási idővel jön létre, mint a ‘.py’ fájl. Ha a két fájl utolsó módosítási ideje különbözik, a ‘.py’ fájlt újrafordítja a Python.

Rendszerint nem kell tenni semmit a ‘spam.pyc’ fájl létrehozásáért, mert az a ‘spam.py’ sikeres bájtkódra fordításakor automatikusan létrejön.

Nem hiba, ha ez az automatizmus látszólag nem működik. Ha bármi oknál fogva a ‘spam.pyc’ írása félbeszakad, a fájl-t a Python később mindenképpen érvénytelen fájl-nak fogja felismerni. A lefordított ‘.pyc’ fájl tartalma platform független, ezért a Python modulkönyvtárat nyugodtan megoszthatod különböző achitektúrájú gépek között.

Néhány tipp haladóknak:

- Amikor a Python értelmezőt a **-O** paraméterrel hívjuk meg, az optimalizált kódot generál és tárol a ‘.pyo’ fájlokban. Az optimalizáló jelenleg nem túl sok segítséget nyújt – egyszerűen csak eltávolítja az `assert` utasításokat. Amikor a **-O** paramétert használjuk, *minden* bájtkód optimalizált lesz. A `.pyc` fájlokat az értelmező figyelmen kívül hagyja, és a `.py` fájlokat optimalizált bájtkódra fordítja.
- Amikor két **-O** paraméterrel hívjuk meg a Python értelmezőt, (**-OO**) a bájtkód-fordító úgy optimalizál, hogy az optimalizáció ritka esetekben hibás programműködést okoz??? ???optimizations that could in some rare cases result in malfunctioning programs.

  Jelenleg csak a `__doc__` szövegeket törli a bájtkódból, tömörebb ‘.pyo’ fájlt eredményezve. Mivel néhány program számít a dokumentációs változó elérhetőségére, csak akkor használd ezt az opciót, ha pontosan tudod hogy mit csinálsz.
- A program semmivel sem fut gyorsabban, ha ‘.pyc’ vagy ‘.pyo’ kiterjesztésű bájtkódot futtatunk – a sebességnövekedés a betöltési időben jelentkezik. Ha a ‘.py’ fálj bájtkódra fordított verziója rendelkezésre áll, nincs szükség a fordításra, rögtön lehet futtatni a bájtkódot tartalmazó verziót.
- Amikor egy szkriptet a parancssorból futtatunk a nevének megadásával, a futó program bájtkódja soha nem íródik ‘.pyc’ vagy ‘.pyo’ fájlba. Ebből kifolyólag a program indulási ideje lerövidíthető, ha a kód nagy részét modulokba helyezzük át, és az indítóprogramunk kis méretű, és importálja a szükséges modulokat. ??? Lehetőség van a ‘.pyc’, vagy ‘.pyo’ fájlok közvetlen futtatására is.???

- Lehetséges, hogy van egy ‘spam.pyc’ (vagy ‘spam.pyo’ fájlod, ha a **-O** fordítási paramétert használták) – úgy, hogy nincs mellette ‘spam.py’ fájlod. Ez egy módja a Python könyvtárak közzétételének úgy, hogy a könyvtárat nem tudják módosítani, megváltoztatni (hiszen nincs meg a forráskód, amiből fordították).
- A `compileall` modul ‘.pyc’ fájlokat tud készíteni az aktuális könyvtár összes moduljából. (vagy ‘.pyo’-t, ha a **-O** kapcsolót használtad)

6.2. Standard modulok

A Python funkciójuk szerint csoportokba sorolt szabványos modulokkal rendelkezik – egy könyvtárral – részletesen: *Python Library Reference – Python referenciakönyvtár a későbbiekben A modulok tételes felsorolása – Module index* Néhány modult az értelmezőbe építettünk be, ezeken keresztül olyan funkciók megvalósítása lehetséges, amelyek ugyan nem tartoznak szorosan a nyelvhez, de például az operációs rendszerrel való kapcsolathoz szükségesek – ilyenek például a rendszerhívások.

Ezen modulok függenek a használt operációs rendszertől, hiszen annak működtetéséhez kellenek. Például az `amoeba` modul csak azokon a rendszereken elérhető, amik támogatják az Amoeba primitívek használatát. A másik figyelemre méltó – és különleges modul a `sys` , ami minden Python értelmezőbe be van építve. Például a `sys.ps1` és `sys.ps2` változók tartalmazzák az értelmezőben megjelenő elsődleges és másodlagos prompt karakterláncát:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Ez a két változó csak akkor létezik, ha az értelmező interaktív módban fut.

A `sys.path` változó karakterláncok listáját tartalmazza, melyek meghatározzák az értelmező keresési útvonalát, amit az a modulok importálásakor bejár. Kezdeti értékét a PYTHONPATH környezeti változóból veszi, vagy ha ez nem létezik, akkor az értelmezőbe beépített alapértelmezett útvonalakból. A változó értékét ugyanúgy módosíthatod, mint egy listáét:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. A `dir()` függvény

A beépített `dir()` függvénnyel listázhatjuk ki a modulban definiált neveket. A `dir()` meghívása után a nevek rendezett listájával tér vissza.

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Ha paraméterek nélkül hívjuk meg a `dir()` függvényt, az aktuális névtérben definiált nevekkel tér vissza:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Fontos, hogy az így kapott lista az összes névfajtát tartalmazza. Változókat, modulokat, függvényeket, stb.

A `dir()` nem listázza ki a nyelvben előre definiált (beépített) függvényeket és változókat. Ezek a `__builtin__` modulban vannak definiálva:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeError', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__debug__', '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
 'setattr', 'slice', 'staticmethod', 'str', 'string', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4. A csomagok

A csomagok adnak lehetőséget a Python modulok névtereinek struktúrálására, a pontozott modulnevek használatával. Például a `A.B` modulnév hivatkozik a ‘`B`’ modulra, ami az ‘`A`’ modulban található (ott importáltuk). Ha a programozók a fenti példa szerint használják a modulokat, nem kell amiatt aggódniuk, hogy egymás globális neveivel ütközés lép fel. Például a több modulból álló csomagok (NumPy, Python Imaging Library...) írói is a pontozott modulnevek használatával kerülik el a változónevek ütközését.

Tegyük fel, hogy egy modulokból álló csomagot akarsz tervezni, hogy egységesen tudd kezelni a hangfájlokat és a bennük lévő adattartalmat. Több különböző hangfájlformátum létezik (rendszerint a kiterjesztésük alapján lehet őket beazonosítani, pl.: ‘.wav’, ‘.aiff’, ‘.au’) – valószínűleg egy bővülő modulcsoportot kell készítened és karbantartanod a fájlformátumok közötti konvertálásra.

There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

Ráadásul még többfajta műveletet is el kell tudnod végezni a hanganyagon, például keverést, visszhang készítését, hangszínszabályzást, művészeti sztereo effekteket – szóval a fentiek tetejébe még írni fogsz egy végeláthatatlan modulfolyamot, ami ezeket a műveleteket elvégzi. A csomagok egy lehetséges struktúrája – a hierarchikus fájlrendszereknél használatos jelöléssel:

```
Sound/                          Legfelső szintű csomag
      __init__.py               a sound csomag inicializálása
      Formats/                  A fájlformátum konverziók alcsomagja
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  A hangeffektek alcsomagja
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  A szűrők alcsomagja
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

A csomag: olyan hierarchikus könyvtárszerkezet, amely egymással összefüggő modulokat tartalmaz.

Egy csomag importálása során a Python bejárja a `sys.path` változóban szereplő könyvtárakat, a csomag alkönyvtárak után kutatva. A `sys.path` az előre meghatározott keresési útvonalakat tartalmazza.

A Python az ‘__init__.py’ fájlok jelenlétéből tudja, hogy egy könyvtárat csomagként kell kezelnie – és ez a fájl segít abban is, hogy az alkönyvtárakban lévő csomagokat is érzékelje a Python.

A legegyszerűbb esetben az ‘__init__.py’ egy üres fájl, de tartalmazhat és végrehajthat a csomaghoz tartozó inicializáló kódot, vagy beállíthatja az `__all__` változót (lásd lejjebb).

A csomag felhasználói egyenként is importálhatnak modulokat a csomagból:

```
import Sound.Effects.echo
```

Ez betölti a `Sound.Effects.echo` almodult. A hivatkozást teljes útvonallal kell megadni.

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Egy másik alternatíva almodulok importálására:

```
from Sound.Effects import echo
```

Ez szintén betölti az `echo` almodult, és elérhetővé teszi a csomagnevek nélkül is (nem kell a Sound.Effects előtag).

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Van még egy lehetőség a kiválasztott függvény importálására:

```
from Sound.Effects.echo import echofilter
```

Ez szintén betölti az `echo` almodult, de a `echofilter()` függvény közvetlenül elérhetővé válik:

```
echofilter(input, output, delay=0.7, atten=4)
```

Fontos, hogy a `from` *csomag* `import` *elem* használatakor az elem az importált csomag almodulja (submodule) vagy alcsomagja (subpackage) is lehet, vagy valamilyen más, a csomagban definiált 'elem' nevű objektum, például függvény, osztály vagy változó.

Az `import` utasítás először ellenőrzi, hogy az importálandó elem definiálva van-e a csomagban. Ha nem, akkor az elemről feltételezi, hogy az egy modul, és megkísérli azt betölteni. Ha a modul keresése sikertelen, `ImportError` kivétel váltódik ki.

Ezzel ellentétben az `import` *elem.alelem.alelem_aleleme* utasításforma használatakor az utolsó alelem_aleleme kivételével mindegyik elemnek csomagnak kell lennie. Az utolsó elem lehet modul vagy csomag is, de a fentiekkel ellentétben nem lehet osztály, függvény vagy definiált változó.

6.4.1. Egy csomag összes elemének importálása

Mi történik, amikor a programozó kiadja a `from Sound.Effects import *` utasítást? Ideális esetben az értelmező a fájlrendszerben megtalálja a csomagban lévő almodulokat, és mindet importálja.

Sajnos ez a művelet Mac vagy Windows platformon nem igazán jól működik – ennek oka a fájlrendszerek hiányosságában keresendő, hiszen az nem tartalmaz pontos információkat a fájlnév pontos kisbetűs - nagybetűs írásmódjáról. A fenti platformokon nincs garantált módszer annak kiderítésére, hogy az 'VISSZHANG.PY' fájlt `visszhang`, `Visszhang` vagy `VISSZHANG` modulként kell-e importálni.

Például a Windows 95 egyik bosszantó tulajdonsága, hogy minden fájlnevet nagy kezdőbetűvel jelenít meg. A DOS 8+3 betűs névhosszúsága szintén érdekes problémákat vet fel hosszú modulnevek esetében.

A csomagkészítők számára az egyedüli megoldás az, ha a csomagot egyértelmű index-el látják el. Az import utasítás a következő szabályokat használja: ha a csomag '`__init__.py`' állományának kódjában szerepel az `__all__` nevű lista, az abban felsorolt nevek lesznek az importálandó modulnevek a `from` *package* `import *` utasítás végrehajtásakor.

A csomag készítőjének feladata, hogy ezt a listát naprakészen tartsa, mikor a csomag újabb verzióját készíti. A csomagkészítők dönthetnek úgy, hogy ezt a funkciót nem támogatják, ha valószínűtlennek tartják hogy valaki az `import *` utasítást használja a csomagra. Például a 'Sounds/Effects/__init__.py' fájl a következő kódot tartalmazhatja:

```
__all__ = ["echo", "surround", "reverse"]
```

A fentiek értelmében a `from Sound.Effects import *` importálni fogja a három felsorolt almodult a `Sound` csomagból.

Ha az `__all__` lista nem meghatározott, a `from Sound.Effects import *` utasítás *nem* importálja a `Sound.Effects` csomag összes almodulját az aktuális névtérbe – csupán azt biztosítja, hogy a `Sound.Effects` csomag importálva legyen (talán, ha az '__init__.py' inicializáló kódból adod ki az utasítást ???) és aztán a csomagban található összes nevet importálja.

Ebbe beleértendő minden név, amit az '__init__.py'-ben vagy az almoduljaiban definiáltak. Szintén importálva lesz a csomag minden almodulja, amit a `from Sound.Effects import *` utasítás előtt importáltunk.

Figyeljük meg, mi történik ebben a kódban:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Ebben a példában az 'echo' és a 'surround' modulokat az értelmező a helyi (belső) névtérbe importálja, mert a `Sound.Effects` csomag részei voltak a `from...import` utasítás végrehajtásakor. (Ez szintén működik, amikor az `__all__` változó definiálva van)

Jó tudni, hogy az `import *` importálási mód használata kerülendő, mert a kódot nehezen olvashatóvá teszi. Ámbár ennek az importálási módnak a használatával egyszerűbben dolgozhatunk az értelmező interaktív módjában, egyes modulokat úgy terveznek, hogy csak az `__all__` változóban megadott neveket exportálják.

Emlékezzünk rá, hogy semmi probléma nincs a `from Package import specific_submodule` szerkezet használatával! Valójában ez az ajánlott importálási mód, hacsak az importáló modulnak nincs szüksége különböző csomagokból származó azonos nevű almodulok használatára.

6.4.2. Csomagon belüli hivatkozások

Az almodulok gyakran hivatkoznak egymásra – például a `surround` modul elképzelhető hogy használja az `echo` modult.

Valójában ha a példához hasonló hivatkozás történik, az `import` utasítás először a már importált csomagokban keres, mielőtt a standard modulok keresési útvonalát végignézi.

Ennélfogva a surround modul egyszerűen használhatja az `import echo` vagy a `from echo import echofilter` importálási utasítást. Ha az importált modul nem található meg az aktuális csomagban (az a csomag, amelyiknek az aktuális modul is része – almodulja), az `import` utasítás a felső szintű modulokban keresi a megadott nevet.

Amikor a csomagok kisebb egységekből – alcsomagokból épülnek fel, a fenti példában szereplő `Sound` csomaghoz hasonlóan, az almodulok egymásra teljes – pontozott, hierarchikus nevükkel kell hivatkozzanak. Például ha a `Sound.Filters.vocoder`-nek használnia kell a `Sound.Effects` csomagban található `echo` modult, a következőképpen érheti azt el: `from Sound.Effects import echo`

6.4.3. Modulok, amelyek több, különálló könyvtár moduljaiból épülnek fel

A csomagok rendelkeznek egy egyedi tulajdonsággal, melyet `__path__` -nak hívunk. Ez egy lista, amelyben az '__init__.py' fájlt tartalmazó könyvtár nevét találjuk – mielőtt az aktuális fájlban lévő kód végrehajtódna.

Ez egy módosítható változó, amely befolyásolja a csomagban található modulok és alcsomagok keresését. Bár erre a lehetőségre ritkán van szükség, a csomagot újabb modulokkal egészíthetjük ki vele.

7. fejezet

Bemenet és kimenet (2.4 doc)

Egy program kimenete többféleképpen jelenhet meg: például ember által olvasható nyomtatott formában, vagy későbbi feldolgozás céljából fájlba írva. Ez a fejezet a lehetőségek közül többet is bemutat.

7.1. Esztétikus kimenet kialakítása

Az értékek kiírására két lehetőségünk van: *Saját kimenet-formázás függvényekkel ???* és a `print` utasítás. (a fent említett harmadik lehetőség a `write()` metódust használja fájl objektumok kezelésére; a szabványos kimeneti fájlt a `sys.stdout` -ként érhetjük el. A referencia könyvtárt érdemes megnézni (Library Reference), ott több információt találunk a szabványos kimenet használatáról.

Gyakori igény, hogy a programozó az egyszerű - szóközökkel tagolt kimeneti formázásnál több lehetőséget szeretne. A kétféleképpen formázhatod a kimenetet:

- A kimeneti karakterláncok teljes formázását te kezeled – a karakterláncok szétvágásával és összeillesztésével bármilyen elképzelt kimenetet előállíthatsz. A szabváyos modulok között található `string` számos hasznos karakterlánc-kezelő rutint tartalmaz (pl. karakterlánc kitöltése adott szélesség eléréséhez) – ezek rövid bemutatása lejjebb található.
- A második lehetőség a % operátor használata, ahol a formázandó karakterlánc baloldali argumentum (Példa pár sorral lentebb!). A % operator feldolgozza a baloldali karakterláncot, mintha az `sprintf()` függvényt használtuk volna – és a formázott karakterláncot adja visszatérési értékként.

Egy kérdés maradt hátra: hogy hogyan konvertálhatsz egy számjegyekből álló karakterláncot értékkel bíró számmá? Szerencsére a Python erre több lehetőséget is biztosít: add át a karakterláncot a `repr()` vagy az `str()` függvényeknek. A fordított egyszeres idézőjelek (‘‘) használata egyenértékű a `repr()` függvény hívásával, de használatuk helyett a függvényhívást javasoljuk.

Az `str()` függvény az értékek ember által olvasható formájával tér vissza, míg a `repr()` függvény visszatérési értéke az interpreter számára értelmezhető adat. (vagy `SyntaxError` kivételt vált ki nem megfelelő szintaxis esetén.)

Azon objektumok esetében, amelyeknek nincs emberi olvasásra szánt ábrázolása, az `str()` függvény ugyanazzal az értékkel tér vissza, mintha a `repr()` függvényt hívtuk volna meg. Néhány érték, például a számok vagy a struktúrák esetében – mint például a listák vagy a szótárak, bármely két fent említett funkciót használva ugyanazt az eredményt kapjuk. Karakterláncok és lebegőpontos számoknak két eltérő ábrázolási módjuk van:

Néhány példa:

```
>>> s = 'Helló, világ.'
>>> str(s)
'Helló, világ.'
>>> repr(s)
"'Helló, világ.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x értéke: ' + repr(x) + ', és y értéke: ' + repr(y) + '...'
>>> print s
x értéke: 32.5, és y értéke: 40000...

% >>> # The repr() of a string adds string quotes and backslashes:
>>> # A repr() függvény idézőjelek közé rakja a stringet,
>>> # és kijelzi a különleges (escape) karaktereket is:
... hello = 'helló világ\n'
>>> hellos = repr(hello)
>>> print hellos
'helló világ\n'

>>> # A repr() függvénynek akár objektumokat is átadhatunk:
... repr((x, y, ('hús', 'tojás')))
"(32.5, 40000, ('hús', 'tojás'))"

% >>> # reverse quotes are convenient in interactive sessions:
>>> # A visszahajló idézőjeleket (‘) interaktív módban használhatjuk,
>>> # ugyanazt érjük el, mint ha a repr() függvényt hívtuk volna meg:
... ‘x, y, ('spam', 'eggs')‘
"(32.5, 40000, ('spam', 'eggs'))"
```

Ha akarunk készíteni egy táblázatot, amiben a számok második és harmadik hatványai szerepelnek, két lehetőségünk is van:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
%...     # Note trailing comma on previous line
...     # Figyelem! az előző sor végén szereplő vessző miatt a
...     # következő print az előző sor végén folytatja a kiírást!
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Megjegyzés: az oszlopok között 1 szóköznyi helyet a `print` utasítás hagyott – az utasítás a paraméterei között mindig 1 szóközt hagy.)

Ez a példa bemutatja a szöveges (karakterlánc) objektumok `rjust()` metódusát, ami a megadott karakterláncot jobbra igazítja, majd a megadott szélességig feltölti üres karakterekkel a baloldalt. Ehhez hasonlóak a `ljust()` és a `center()` függvények. Ezek írási műveletet nem végeznek, egyszerűen visszatérnek az új karakterlánccal. Ha a bemenetként megadott szöveg túl hosszú, azt nem csonkolják – változatlanul adják vissza az eredeti karakterláncot. Ez elrontja ugyan a kimenet rendezettségét, de rendszerint jobb, mintha a függvény valótlan (csonkított) értékkel térne vissza. (Ha valóban szeletelni akarod a karakterláncot, ezt így tudod megtenni: '`x.ljust( n)[:n]`'.)

Létezik egy másik metódus, a `zfill()`, amely az adott numerikus karakterláncot balról nulla karakterekkel tölti fel. Ez könnyen megérthető a plusz és minusz jelek esetében:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

A % operátort használata így néz ki:

```
>>> import math
>>> print 'PI értéke megközelítőleg %5.3f.' % math.pi
PI értéke megközelítőleg 3.142.
```

Ha a karakterláncban egynél több formázást szeretnél használni, akkor az alábbi példát követve egy tuple változót

kell paraméterként használnod:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

A legtöbb formázás pontosan ugyanúgy működik, mint C-ben, és a megfelelő típusú változó átadását igényli. Ha kivétel váltódik ki, és azt a kódodban nem kapod el, ???not a core dump???.

A `%s` formázás ennél rugalmasabb: ha a csatolt paraméter nem egy karakterlánc, akkor a `str()` beépített függvény használatával automatikusan azzá konvertálja. A `*` használata a szélesség, vagy a pontosság meghatározására egy külön paraméterként (integer) lehetséges. A következő C formázások nem támogatottak: `%n`, `%p`.

Ha egy nagyon hosszú formázott karakterláncot szeretnél használni, amit nem akarsz darabokra felosztani, szép lenne, ha hivatkozni tudnál a formázandó változókra azok neveivel, pozíciójuk helyett. Ezt a `%(name)format` forma használatával teheted meg, például így:

```
>>> table = {'Bea': 1975, 'Balazs': 1978, 'Fanni': 2003}
>>> print 'Fanni: %(Fanni)d; Bea: %(Bea)d; Balazs: %(Balazs)d' % table
Fanni: 2003; Bea: 1975; Balazs: 1978
```

Ez különösen hasznos az új `vars()` függvénnyel együtt, amely egy szótár típusú változóval tér vissza, amely az összes helyi változót tartalmazza.

7.2. Fájlok írása és olvasása

Az `open()` függvény egy object objektummal tér vissza, és rendszerint két paraméterrel használjuk: '`open(`*filename*`, `*mode*`)`'.

```
>>> f=open('/tmp/munkafile', 'w')
>>> print f
<open file '/tmp/munkafile', mode 'w' at 80a0960>
```

Az első paraméter egy fájlnevet tartalmazó karakterlánc. A második paraméter néhány karakterből áll csupán, és a fájl használatának a módját (írás, olvasás) állíthatjuk be vele. A megnyitás módja (*mode*) lehet `'r'` mikor csak olvassuk a fájlt – `'w'`, ha kizárólag írni szerenténk (a már esetleg ugyanezen néven létező fájl tartalma törlődik!) – és `'a'`, amikor hozzáfűzés céljából nyitjuk meg a fájlt; ilyenkor bármilyen adat, amit a fájlba írunk, automatikusan hozzáfűződik annak a végéhez. A `'r+'` megnyitási mód egyszerre nyitja meg a fájlt írásra és olvasásra. A *mode* paraméter beállítása nem kötelező; elhagyása esetén `'r'` alapértelmezett értéket vesz fel.

Windows és Macintosh rendszereken a megnyitási módhoz hozzáfűzött `'b'` karakterrel bináris módban nyithatjuk meg a fájlt, például így: `'rb'`, `'wb'` vagy `'r+b'`. A Windows megkülönbözteti a szöveges és bináris fájlokat; a szöveges fájlban a sorvéget jelző karakterek (end-of-line) kis mértékban automatikusan megváltoznak adat írása vagy olvasása esetén.

Ez a háttérben zajló módosítás ugyan megfelelő az ASCII szöveges fájlok számára, de a bináris fájlokat használhatatlanná teszi (pl. 'JPG' vagy '.EXE' fájlokat). Ezért nagyon figyelj oda, hogy a bináris módot használd olvasáskor és íráskor. (Megjegyezzük, hogy Macintosh-on a használt C könyvtártól függ a megnyitási módok pontos működése.

7.2.1. A fájl objektumok metódusai

A fejezet további példái feltételezik, hogy már létezik az `f` fájl objektum.

A fájl tartalmának olvasásához hívd meg az `f.read(`*size*`)` metódust, ami a megadott adatmennyiségnek megfelelő hossszúságú karakterlánccal tér vissza. A *size* egy opcionális paraméter – elhagyása, vagy negatív értéke esetén a teljes tartalmat visszaadja a metódus – ha esetleg a fájl kétszer akkora, mint a gépedben lévő memória, az esetleg problémát jelenthet neked.

Ha használod a *size* paramétert, akkor a visszatérési értékként kapott karakterlánc hosszát maximalizálni tudod. Ha eléred a fájl végét, az `f.read()` egy üres karakterlánccal tér vissza (`""`).

```
>>> f.read()
'Ez a fájl teljes tartalma.\n'
>>> f.read()
''
```

A `f.readline()` egy sort olvas ki a fájlból. A sor végét az újsor karakter (`\n`) jelenti, amely a beolvasott karakterlánc végén található. Ez a karakter egyetlen esetben maradhat ki a visszaadott karakterlácból: ha a fájl utolsó sorát olvassuk be, és az nem újsor karakterre végződik.

Ez a visszatérési értéket egyértelművé teszi: ha a `f.readline()` metódus üres karakterlánccal tér vissza, az olvasás elérte a fájl végét. Ekkor az üres sort a `'\n'` karakter jelképezi – a karakterlánc egyetlen egy újsor karaktert tartalmaz.

```
>>> f.readline()
'Ez a fájl első sora.\n'
>>> f.readline()
'A fájl második sora.\n'
>>> f.readline()
''
```

A `f.readlines()` metódus egy listával tér vissza, amely a fájl minden sorát tartalmazza. Ha megadjuk a *sizehint* paramétert, a metódus a megadott számú byte-ot kiolvassa a fájlból, és még annyit, amennyi a következő újsor karakterig tart. (A fordító megjegyzése: *sizehint* paramétert hiába adtam meg, 2.1-es pythont használva a teljes fájltartalmat kiolvasta.)

```
>>> f.readlines()
['Ez a fájl első sora.\n', 'Ez pedig a második\n']
```

A `f.write(`*string*`)` metódus a *string* tartalmát a fájlba írja, és `None` értékkel tér vissza.

```
>>> f.write('Tesztszöveg, az írás bemutatására\n')
```

Ha egy karakterlánctól eltérő típusú változót szeretnénk kiírni, akkor azt előbb karakterlánccá kell konvertálni:

```
>>> value = ('a valasz', 42)
>>> s = str(value)
>>> f.write(s)
```

Ford.: Ha megnézzük a keletkezett fájl tartalmát, az `('a valasz', 42)` lesz.

Az `f.tell()` metódussal a fájl objektum aktuális pozícióját kérdezhetjük le – bájt-ban, a fájl kezdetétől számolva. (pl.: hányadik bájt-ot olvassuk most éppen)

A fájl objektum pozíciójának (a kurzornak) a megváltoztatásának módja: '`f.seek(`*léptetés*`, `*innen_kezdve*`)`'. Az *innen_kezdve* ponttól *léptetés* mennyiséggel mozgatjuk a kurzort. (Példa: ha van egy 100 bájtos fájl, amiben éppen a 39. karakternél állunk, akkor az aktuális pozícióhoz képest ugorhatunk tovább, megadott lépésekben)

A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

Az *innen_kezdve* paraméter a következő értékeket veheti fel:

- 0: a fájl eleje lesz a hivatkozási pont
- 1: az aktuális pozíció lesz a hivatkozási pont
- 2: a fájl vége az aktuális hivatkozási pont

```
>>> f = open('/tmp/munkafajl', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Ugorj a 6. bajthoz a fajlban
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Ugorj a fajl vegehez kepest harom karakterrel vissza
>>> f.read(1)
'd'
```

Ha már minden módosítást elvégeztél a fájl-al, hívd meg a `f.close()` metódust a fájl bezárásához (a rendszererőforrások felszabadításához). A `f.close()` hívása után a fájl objektum használatára tett kísérletek automatikusan meghiúsulnak.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

A fájl objektumoknak van néhány kiegészítő metódusuk, például az `isatty()` és a `truncate()` – ezek ritkábban használtak. A Referencia könyvtárban teljes útmutatót találsz a fájl objektumok használatához.

7.2.2. A `pickle` modul

A karakterláncokat könnyű fájlba írni és onnan kiolvasni. A számokkal kicsit nehezebb a helyzet, mert a `read()` metódus mindent karakterként ad vissza, amit aztán át a számmá történő átalakítás miatt például az `int()` függvénynek kell átadnunk, ami a `'123'` karakterláncból a 123-as értéket adja vissza.

Hogyha összetettebb adattípusokat akarsz menteni, például listákat, szótárakat vagy osztály-példányokat, egyre bonyolultabbá válik a dolgod.

Ahelyett, hogy a felhasználóknak állandóan adatmentő algoritmusokat kelljen írnia és javítania, a Python rendelkezik a `pickle` modullal. Ez egy elképesztő modul, ami képes a legtöbb Python objektumot karakterláncként ábrázolni. (even some forms of Python code!???) Ezt a folyamatot *pickling*-nek hívják.

Az objektum karakterláncból történő létrehozását pedig *unpickling*-nek nevezik.

A karakterlánccá történő konvertálás és a visszakonvertálás között a karakterlánc jelképezi az objektumot. Ezt akár fájlban, akár más formában tárolhatjuk. (hálózaton elküldve másik gépen, vagy adatbázisban)

Ha van egy `x` objektumod, és létezik az írásra megnyitott `f` fájl objektum, a legegyszerűbb út az objektumod tárolására a következő egysoros kód:

```
pickle.dump(x, f)
```

Ha újból elő kívánod állítani az objektumot, és az `f` egy olvasásra megnyitott fájl objektum:

```
x = pickle.load(f)
```

(Vannak más variációi is ennek, amik több objektum tárolásánál / újbóli előállításánál használatosak, illetve akkor, ha nem akarod a tárolási formába alakított objektumodat ténylegesen fájlba írni; a téma teljes dokumentációja: `pickle` a *Python Library Reference*-ben.)

A `pickle` a szabályos módja a Python objektumok készítésének, amiket tárolhatsz, vagy újra felhasználhatsz más programokkal, vagy akár a jövőbeni felhasználás céljából a jelenlegi programoddal. Ennek a technikai megnevezése: *persistent* objektumok (állandó, tartós). Azért, mert a `pickle` széles körben elterjedt, a legtöbb Python kiegészítést készítő programozó figyelmeztet, hogy bizonyosodj meg arról, hogy az új adattípusok tárolhatók és újra előállíthatók (mintha egy matricát leragasztanál, majd újra felvennéd).

8. fejezet

Hibák és kivételek

Eddig csak említettük a hibajelzéseket, de ha kipróbáltad a példákat feltehetően láttál néhányat. Kétfajta megkülönböztetendő hibafajta van (legalább): *szintaktikai hiba (syntax error)* és *kivétel (exception)*.

8.1. Szintaktikai hibák

A szintaktikai hibák, vagy másképp elemzési hibák, talán a leggyakoribb hibaüzenetek, amíg tanulod a Pythont:

```
>>> while 1 print 'Hello világ'
  File "<stdin>", line 1
    while 1 print 'Hello világ'
                ^
SyntaxError: invalid syntax
```

Az elemző megismétli a hibás sort, és kitesz egy kis „nyilat” amely a sorban előforduló legelsőnek észlelt hibára mutat. A hibát a nyilat *megelőző* szócska (token) okozza (vagy legalábbis itt észlelte az értelemező): a példában, a hibát a `print` utasításnál észlelte, mivel a kettőspont (‘`:`’) hiányzik előle. A fájl neve és a sorszám kiíródik, így tudhatod, hol keressed, ha egy szkriptet futtattál.

8.2. Kivételek

Ha egy állítás vagy kifejezés szintaktikailag helyes, akkor is okozhat hibát, ha megpróbáljuk végrehajtani. A végrehajtás során észlelt hibákat *kivételeknek (execptions)* nevezzük és nem feltétlenül végzetesek: nemsokára megtanulod, hogyan kezeld ezeket a Python programban. A legtöbb kivételt általában nem kezelik a programok, ekkor az alábbiakhoz hasonló hibaüzeneteket adnak:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

A hibaüzenetek utolsó sora mutatja, mi történt. Különböző típusú kivételeket kaphatunk, és a típus az üzenet részeként kerül kiíratásra: a típusok a példákban: `ZeroDivisionError` (nullával való osztás), `NameError` (név hiba) és `TypeError` (típussal kapcsolatos hiba). A kivétel típusaként kiírt szöveg a fellépő kivétel belső (built-in) neve. Ez minden belső kivételre igaz, de nem feltétlenül igaz a felhasználó által definiált kivételekre (habár ez egy hasznos megállapodás). A szokásos kivételnevek belső azonosítók (nem fenntartott kulcsszavak).

A sor fennmaradó része egy részletezés, amelynek alakja a kivétel fajtájától függ.

Az ezt megelőző része a hibaüzenetnek megmutatja a szövegkörnyezetet – ahol a kivétel történt – egy veremvisszakövetés (stack backtrace) formájában. Általában ez veremvisszakövetést tartalamazza egy listában a forrás sorait. jóllehet, ez nem fog megjeleníteni a sztenderd bementerről kapott sorokat.

A *Python Library Reference* felsorolja a belső kivételeket és a jelentéseiket.

8.3. Kivételek kezelése

Van rá lehetőség, hogy olyan programokat írjunk, amik kezelik a különböző kivételeket (exception). Nézzük csak a következő példát, amely addig kérdezi a felhasználótól az értéket amíg egy egész számot nem ír be, de megengedi a felhasználónak, hogy megszakítsa a program futását (a `Control-C` használatával, vagy amit az operációs rendszer támogat). Jegyezzük meg, hogy a felhasználó által létrehozott megszakítás `KeyboardInterrupt` kivételként lép fel.

```
>>> while 1:
...     try:
...         x = int(raw_input("Írj be egy számot: "))
...         break
...     except ValueError:
...         print "Ez nem egész szám. Próbáld újra... "
...
```

A `try` utasítás a következőképpen működik.

- Először a *try mellékág* hajtódik végre, azaz a `try` és `except` közötti utasítások.
- Ha nem lép fel kivétel, az *except ág*at a program átugorja, és a `try` utasítás befejeződik.
- Ha kivétel lép fel a try ág végrehajtása során, az ág maradék része nem hajtódik végre. Ekkor – ha a típusa megegyezik az `except` kulcsszó után megnevezett kivétellel, ez az expect ág hajtódik végre, és ezután a végrehajtás a try–except blokk után folytatódik.
- Ha olyan kivétel lép fel, amely nem egyezik a az expect ágban megnevezett utasítással, akkor a kivételt átadja egy külsőbb `try` utasításnak. Ha nincs kezelve a kivétel, akkor ez egy *kezeletlen kivétel* a futás megáll egy utasítással, ahogy korábban láttuk.

A `try` utasításnak lehet egynél több except ága is, hogy a különböző kivételeket kezelhessük. Egynél több kezelő hajtódhat végre. A kezelők csak a hozzájuk tartozó try ágban fellépő kivételt kezelik a `try` utasítás másik kezelőjében fellépő kivételt nem. Egy expect ágat több névvel is illethetünk egy kerek zárójelbe tett lista segítségével, például:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

(Magyarul: Futásidejű hiba, TípusHiba, NévHiba)

Az utolsó expect ág esetén elhagyható a kivétel neve. Rendkívül óvatosan használjuk, mert elfedhet valódi programozási hibákat! Arra is használható, hogy kiírjunk egy hibaüzenetet, majd újra kivételdobást hajtsunk végre, (megengedve a hívónak, hogy lekezelje a kivételt):

```
import string, sys

try:
    f = open('file.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Nem képes az adatot egész számmá alakítani."
except:
    print "Váratlan hiba:", sys.exc_info()[0]
    raise
```

A `try` … `except` utasításnak lehet egy *else ága* is, amelyet – ha létezik – az összes except ágnak meg kell előznie. Ez nagyon hasznos olyan kódokban, amelyeknek mindenképpen végre kell hajtódniuk, ha a try ágban nem lép fel kivétel. Például:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'nem nyitható meg', arg
    else:
        print arg, len(f.readlines()), 'sorból áll.'
        f.close()
```

Az `else` ág használata jobb, mintha a kódot a `try`ághoz adnánk, mivel ez elkerüli egy olyan kivétel elkapását, amely nem a `try` … `except` utasítások által védett ágban vannak.

Ha kivétel lép fel, lehetnek hozzátartozó értékei, amelyeket a kivétel *argumentumának* is nevezünk. A megléte és a típusa a kivétel fajtájától is függ. Azokra a kivételtípusokra, amelyek argumentummal rendelkeznek, az except ág előírhat egy változót a kivétel neve (vagy a lista) után, amely felveszi az argumentum értékét, ahogy itt látható:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'A(z)', x, 'név nincs definiálva.'
...
A(z) spam név nincs definiálva.
```

Ha a kivételnek argumentuma van, az mindíg utolsó részeként kerül a képernyőre.

A kivételkezelők nem csak akkor kezelik a kivételeket, ha azok ténylegesen a try ágban szerepelnek, hanem akkor is, ha azok valamelyik try ágban meghívott függvényben szerepelnek (akár közvetve is). Például:

```
>>> def ez_rossz():
...     x = 1/0
...
>>> try:
...     ez_rossz()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo
```

8.4. Kivételek létrehozása

A `raise` utasítás lehetővé teszi a programozó számára, hogy egy új, általa megadott kivételt hozzon létre. Például:

```
>>> raise NameError, 'IttVagyok'
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: IttVagyok
```

A `raise` első argumentuma a kivétel neve amit létrehozunk. Az esetleges második argumentum adja meg a kivétel argumentumát.

8.5. User-defined Exceptions

A programok elnevezhetik a saját kivételeiket, ha karakterláncot rendelnek egy változóhoz, vagy egy új kivételosztályt hoznak létre. Például:

```
>>> class MyError:
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return 'self.value'
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'A kivételem fellépett, értéke:', e.value
...
A kivételem fellépett, értéke: 4
>>> raise MyError, 1
Traceback (most recent call last):
  File "<stdin>", line 1
__main__.MyError: 1
```

Sok – a Pythonban megtalálható – modul ezt használja a függvényekben előforduló esetleges hibák megjelenítésére.

Bővebb információ az osztályokról a 9 fejezetben található.

8.6. Takarító-lezáró műveletek definiálása

A `try` utasításnak van egy másik opcionális ága, mely takarító-rendberakó műveletek tárolására szolgál – ezeket a megelőző ágak lefutása után kell végrehajtani. Például:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Viszlát világ!'
...
Viszlát világ!
Traceback (most recent call last):
  File "<stdin>", line 2
KeyboardInterrupt
```

A *végső záradék* akár kivételdobás történt a `try` záradékban, akár nem – mindenképpen végrehajtódik. Ha kivételdobás történt, a kivétel a `finally` záradék végrehajtása után újra kiváltódik. A finally – a kivételdobás utószeleként, lefutása végeként – hajtódik végre, amikor a `try` utasítás elhagyja a `break` vagy a `return` utasítást. (Fontos: függvényeknél ha nem helyezünk el `return` utasítást, akkor rejtetten egy `return None` utasítás fut le – tehát ha nem látunk egy utasítást, az esetleg rejtetten akkor is lefuthat!)

A `try` utasítás vagy except záradékok(at) tartalmazhat, vagy egy finally záradékot, de a kettőt együtt nem.

9. fejezet

Osztályok

A Python osztálymechanizmusa osztályokat ad a nyelvhez a lehető legkevesebb szintaxissal és szemantikával. A Python osztályok a C++ és a Modula-3 –ban található osztálymechanizmusok keveréke. Ahogy a modulokra is igaz, az osztályokban a Python nem határolja el élesen a definíciót és a felhasználót: megbízik a felhasználó ítélőképességében, hogy nem akar a "mindent definiálni fogok" hibájába esni.

Az osztályok legfontosabb tulajdonságai teljes egészében megmaradnak: az öröklődési mechanizmus lehetővé teszi a több őstől való származtatást; a származtatott osztályok a szülők bármely metódusát felül tudják írni; a metódusok ugyanazon a néven érhetik el a szülőosztály metódusait. Az objektumok tetszőleges számú egyéni adatot tartalmazhatnak.

C++ szóhasználattal élve az osztály minden eleme (beleértve az adatokat is) *publikus*, és minden tagfüggvény *virtuális*. Nincsenek konstruktor és destruktor függvények. A Modula-3 –hoz hasonlóan nincsen rövidített hivatkozás az objektum alkotóelemeire annak metódusaiból: az objektum függvényeinek deklarálásakor első paraméterként az objektumot jelképező változót adjuk át, melynek értéke értelemszerűen objektumonként változik.

A Smalltalk-hoz hasonlóan az osztályok önmaguk is objektumok, a szó tágabb értelmében, mert a Pythonban minden adattípus objektum.

Ez biztosítja a szó felismerését importáláshoz és átnevezéshez

A C++-tól és a Modula-3 -tól eltérően a beépített típusok szülőosztályokként felhasználhatók. S végül a C++-hoz hasonlóan, de a Modula-3 –tól eltérően a legtöbb, egyéni szintaktikával bíró beépített operátor (aritmetikai műveletek, alprogramok (???: metódusokra gondol?) , stb.) újradefiniálhatók az osztály példányaiban.

9.1. Néhány gondolat a szóhasználatról

Nem lévén általánosan elfogadott szóhasználat az osztályok témakörére, alkalmanként Smalltalk és C++ kifejezéseket fogok használni. (Szerettem volna Modula-3 kifejezéseket alkalmazni, mert annak hasonlít leginkább az objektum-orientáció értelmezése a Pythonhoz, de sajnos nagyon kevesen ismerik ezt a nyelvet.)

Szeretnélek figyelmeztetni a szóhasználatban rejlő csapdára: A Pythonban az objektum szó nem feltétlenül egy osztály példányát jelenti. A C++-hoz és a Modula-3 -hoz hasonlóan, de a Smalltalk-tól eltérően nem minden típus osztály: az alapvető beépített típusok, például az egész számok és a listák (integer, list) nem azok, és néhány eléggé egzotikus típus sem az. (Például a file-ok.) Akárhogy is, *minden* Python típus osztozik az általános jelölésen, amit a legérthetőbben az objektum kifejezés közelít meg.

Az objektumoknak egyéni jellege van, és többszörösen is el lehet nevezni ugyanazt az objektumot (különböző névterekben). Ez a lehetőség fedőnévként (aliasing) ismert más nyelvekben. Ezt a lehetőséget a nyelvvel való első találkozáskor rendszerint nem becsülik meg – egyébként a fedőnév használata nyugodtan mellőzhető megváltoztathatatlan típusok használatakor (például számok, karakterláncok, tuple-ek esetében).

Ugyanakkor a fedőnév használata szándékosan része a Python nyelvtanának a megváltoztatható típusok esetében, mint például a listák, szótárak – és a legtöbb, programon kívüli entitást jelképező típus esetében (file-ok, ablakok, stb.). A fedőnevek általában a program hasznára válnak, különösen amióta a mutatókhoz hasonlítanak néhány vonatkozásban. Például egy objektum átadása kevés erőforrásfelhasználással jár, mióta csak egy mutató (az ob-

jektumra) kerül átadásra a végrehajtás során. Ha a meghívott funkció módosítja a neki átadott objektumot, a hívó látja a változást – ez szükségtelenné teszi két különböző paraméter átadását (mint például a Pascal-ban). (nincs szükség visszatérési értékre a meghívott függvényből, mert a megváltoztatott objektum közvetlenül elérhető a hívó névteréből. A fordító megjegyzése)

9.2. Hatókörök és névterek a Pythonban

Mielőtt megismerkednénk az osztályokkal, beszélnünk kell a hatókörökről. Az osztálydefiníciók néhány ügyes trükköt adnak elő a névterekkel, és neked ismerned kell a névterek és hatókörök működését ahhoz, hogy teljesen átlásd, mi is történik. Egyébként ennek a témakörnek az ismerete minden haladó Python programozónak a hasznára válik.

Kezdetnek nézzünk meg néhány definíciót.

A *névtér* a neveket objektumokra képezi le. A legtöbb névtér jelenleg Python szótárakként van megvalósítva, de ez a teljesítmény kivételével normális esetben nem észlelhető – a jövőben ez egyébként változni fog. Példák a névterekre: a foglalt nevek listája (függvények, például az `abs()`, vagy a beépített kivételek nevei); a modulokban jelenlévő globális nevek; vagy a helyi nevek függvényhívások során. Bizonyos értelemben egy objektum tulajdonságai is külön névteret alkotnak. Fontos tudni, hogy különböző névterekben lévő két név között semmilyen kapcsolat nem létezik. Például ha két különböző modul egyaránt definiálhat egy „maximum_beallitas” nevű függvényt bármilyen következmény nélkül, mert a modulok használóinak a függvénynév előtagjában a modul nevével egyértelműen jelezniük kell, hogy pontosan melyik függvényt fogják használni.

Apropó — a *tulajdonság* jelzőt használom bármilyen névre, ami a pontot követi — például a codez.real kifejezésben a `real` a `z` objektum egyik tulajdonsága.

Az igazat megvallva egy modulbeli névre való hivatkozás egy tulajdonság-hivatkozás: a codemodname.funcname kifejezésben a `modname` egy modul objektum és a `funcname` annak egy tulajdonsága. Ebben az esetben egyenes hozzárendelés történik a modul tulajdonságai és a modulban definiált globális nevek között: ezek egy névtéren osztoznak. [1]

A tulajdonságok lehetnek csak olvashatóak, vagy írhatóak is. Az utóbbi esetben értéket rendelhetünk a tulajdonsághoz – például így: ‘`modname.the_answer = 42`’. Az írható tulajdonságok a `del` utasítással törölhetők is, például a ‘`del modname.the_answer`’ törli a a `modname` objektum `the_answer` tulajdonságát.

A névterek különböző időpontokban születnek, és élettartamuk is változó. Tartalmazzák a Python értelmező beépített neveit, melyek nem törölhetők. A modulok részére a globális névtér a modul definíció olvasásakor jön létre – általános esetben a modul névterek az értelmezőből való kilépésig megmaradnak. Az utasítások az értelmező felső szintű hívásai alapján futnak le, vagy file-ból kiolvasva, vagy az értelmezőbe begépelt utasítások alapján – a `__main__` modul megkülönböztetett ???considered??? részeként, ezért saját névtérrel rendelkeznek. (A beépített nevek szintén a modulban léteznek, `__builtin__` név alatt.).

A függvények helyi névtere a függvény hívásakor keletkezik, és a függvény lefutásakor, vagy le nem kezelt kivételek létrejöttekor szűnnek meg. (Talán a felejtés szó pontosabb jelző lenne a törlés helyett.) Természetesen a rekurzív hívások mindegyike saját, helyi névtérrel rendelkezik.

A *hatókör (scope)* a Python kód azon szöveges része ahol a névtér közvetlenül elérhető. A közvetlen elérhetőség itt azt jelenti, hogy a név a teljes elérési útjának kifejtése nélkül elérhető a névtérben. (például a codez.real-ban a . jelzi, hogy a codez objektumhoz tartozó tulajdonságról van szó, ez itt most teljesen kifejtett.)

Ámbár a névterek meghatározása statikus, dinamikusan használjuk őket. Bármikor a program futása során legalább három közvetlenül elérhető névtér létezik: a belső névtér, amiben az értelmező először keres, és helyi neveket valamint függvények neveit tartalmazza (a függvényeket mindig a legközelebbi zárt névtérben keresi az értelmező) — a másodszor vizsgált középső névtér, ami az aktuális modul globális változóinak neveit tartalmazza — valamint az utoljára vizsgált külső névtér, ami a beépített neveket tárolja.

Ha egy változónév globálisan deklarált, minden hivatkozás és művelet közvetlenül a középső névtérben keres, mert ott találhatók a modul globális nevei. Fontos tudni, hogy a belső névtéren kívül található nevek csak olvashatóak.

[1] Kivéve egy esetet: a modul objektumoknak van egy rejtett csak olvasható tulajdonságuk, a `__dict__` amit kiolvasva megkapjuk a modul névterében lévő nevek listáját egy szótár típusú változóban. A `__dict__` egy tulajdonság, de nem egy globális név. Használata nyilvánvalóan megsérti a névtér-elv alkalmazásának tisztaságát, és csak olyan végső esetekben alkalmazható, mint a program-kiakadások után futtatott hibakeresők.

Rendszerint a helyi névtér a szövegkörnyezetben található helyi változókra hivatkozik az aktuális függvényben. A függvényeken kívül a helyi névtér a globális névtérhez hasonlóan egyben az aktuális modul névtere is. Az osztálydefiníciók pedig újabb névtereket helyeznek el a helyi névtérben.

Tudatosítani kell, hogy a névterek a szövegkörnyezet által meghatározottak: a modulban definiált függvény globális névtere a modul névterében jön létre – a függvény nevei kizárólag itt elérhetők.

Másrészről az aktuális nevek keresése még dinamikusan, futásidőben történik, – a nyelvi definíció akármennyire is törekszik a fordításkori, statikus névfeloldásra, szóval hosszútávon ne számíts a dinamikus névfeloldásra! (Igazság szerint a helyi változók már statikusan meghatározottak)

A Python egy különleges tulajdonsága, hogy a hozzárendelés mindig a belső névtérben történik. A hozzárendelés nem másol adatokat — csak kötést hoz létre a nevek és az objektumok között. A törlésre ugyanez igaz: a '`del x`' utasítás eltávolítja az `x` kötését a helyi névtér nyilvántartásából,

Valójában minden művelet, ami új név használatát vezeti be, a helyi névteret használja – például utasítások importálása, és függvénydefiníciók modulbeli létrehozása vagy kötése.

A `global` kulcsszóval jelezheted hogy bizonyos változók a globális névtérben léteznek.

9.3. Első találkozás az osztályokkal

Az osztályok használatához szükségünk van valamennyi új nyelvtanra, három új objektumtípusra, és néhány új kifejezés használatára.

9.3.1. Az osztálydefiníció szinaxisa

A legegyszerűbb osztálydefiníció így néz ki:

```
class Osztalynev:
    <utasitas-1>
    .
    .
    .
    <utasitas-N>
```

Az osztálydefiníciók hasonlítanak a függvények definíciójára (`def` statements) abból a szempontból, hogy az osztály deklarációjának meg kell előznie az első használatot. Osztálydefiníciót elhelyezhetsz egy `if` utasítás valamely ágában is, vagy egy függvénybe beágyazva.

A gyakorlatban az osztályokon belüli utasítások többsége általában függvénydefiníció, de bármilyen más utasítás is megengedett, és néha hasznos is — erre még később visszatérünk. Az osztályon belüli függvényeknek egyedi argumentumlistájuk (és hívási módjuk) van az osztály metódusainak hívási módja miatt — ezt szintén később fogjuk megvizsgálni.

Egy osztálydefinícióba való belépéskor új névtér jön létre és helyi névtérré válik — ebből kifolyólag minden helyi változóra történő hivatkozás átkerül ebbe az új névtére. ???A gyakorlatban általában az új függvénydefiníciók kerülnek ide.???

Az osztálydefiníciókból való normális kilépéskor egy *class object* objektum jön lére. Ez lényegében egybefoglalja, beburkolja az osztálydefiníciókor létrejött új névtér tartalmát – az osztályobjektumokról a következő alfejezetben fogunk többet tanulni. Az eredeti helyi névtér (az osztálydefinícióba való belépés előtti állapotában) helyreállítódik, és az osztályobjektum neve is a helyi névtér része lesz (az `Osztalynev` a példában).

9.3.2. Osztályobjektumok

Az osztályobjektumok a műveletek kétféle típusát támogatják: attribútum műveletek és a példányosítás. Az *Attribútum tulajdonságok* az általánosan használt Python jelölésmódot használják: `objektum.attribútumnév`.

Az összes név érvényes attribútumnév ami az osztály névterében volt az osztályobjektum létrehozásakor. Ha egy osztály definíció valahogy így néz ki:

```
class Osztalyom:
    "Egy egyszeru pelda osztaly"
    i = 12345
    def f(self):
        return 'hello vilag'
```

akkor codeOsztalyom.i és `Osztályom.f` egyaránt érvényes attribútum hivatkozás – egy egész számmal és egy objektum-eljárással térnek vissza. Az osztályattribútumoknak ugyanúgy adhatunk értéket mint egy normális változónak (`Osztalyom.i = 2`). A `__doc__` eljárás is érvényes attribútum, ami az osztály definíciójában lévő első szabad, nem hozzárendelt string-objektummal tér vissza: `"Egy egyszeru pelda osztaly"`

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

Egy osztály *példányosítása* a függvények jelölésmódjához hasonló. Egyszerűen úgy kell tenni, mintha az osztályobjektum egy paraméter nélküli függvény lenne, amit meghívva az osztály egy új példányát kapjuk visszatérési értékként. Például (a fenti osztályt alapul véve):

```
x = Osztalyom()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

készít egy új *példányt* az osztályból, és hozzárendeli a visszatérési értékként kapott objektumot az `x` helyi változóhoz.

//:: // 2004. okt. 7:

A példányosítás művelete (az objektum „hívása”) egy üres objektumot hoz létre. Elképzelhető, hogy az új példányt egy ismert kezdeti állapotba állítva szeretnénk létrehozni. Ezt egy különleges metódussal, az `__init__()`-el tudjuk elérni:

```
    def __init__(self):
        self.data = []
```

Ha az osztály definiálja az `__init__()` metódust, egy új egyed létrehozásakor annak `__init__()` metódusa automatikusan lefut. Lássunk egy példát egy új, inicializált egyedre:

```
x = Osztalyom()
```

Természetesen az `__init__()` metódusnak paramétereket is átadhatunk. A paraméterek az osztály példányosítása során az inicializáló metódushoz jutnak. Lássunk egy példát:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. A létrehozott egyedek

És most mihez tudunk kezdeni az új egyedekkel? Fontos hogy tisztában legyünk az új objektumok lehetséges alkotóelemeivel. Két lehetséges alkotóelem van: a belső változók (adat tulajdonságok) és az ezekkel dolgozó függvények.

A Pythonban használt *adat tulajdonságok* fogalma megegyezik a Smalltalk „példány változó” fogalmával, és a C++„adat tagok” fogalmával. Az adat tulajdonságokat nem kell a használatuk előtt deklarálni, a helyi változókhoz hasonlatosan működnek – az első használatukkor automatikusan létrejönnek. Például ha `x` az `Osztalyom` egy példánya, a következő kódrészlet 16-ot fog kiírni: ("anélkül, hogy letérnénk az útról", Ford. javítani!)

```
x.szamlalo = 1
while x.szamlalo < 10:
    x.szamlalo = x.szamlalo * 2
print x.szamlalo
del x.szamlalo
```

Fentebb már említettük, hogy az objektumnak lehetnek saját függvényei (más néven metódusai) is. Fontos hogy ezek működését is megértsük. A metódus pontosan egy objektumhoz tartozó függvényt jelöl. (A Pythonban a metódus kifejezés nem kizárólag egy osztály példányát jelenti – más objektum típusok is rendelkezhetnek metódusokkal. Például a lista objektumoknak vannak saját metódusai: append, insert, remove, sort, és így tovább. A ford. megjegyzése: valójában ezek is egy – a nyelvbe beépített osztály példányai, csak tudatosítani kell, hogy itt a Python helyettünk elvégzi a példányosítást, és mi ennek a példánynak a metódusait használjuk. Az alábbi sorokban a metódus kifejezést kizárólag egy osztály metódusaira értjük, hacsak nincs külön kihangsúlyozva, hogy most egy másik objektum metódusáról van szó.

A létrehozott objektum metódusainak neve a szülőosztálytól függ. (Egy osztály belső elemei lehetnek: függvények vagy változók) Meghatározás szerint minden felhasználó által definiált osztályfüggvényt az adott (létező) példány nevével kell hívni. Például `x.f` egy érvényes függvényhivatkozás, ha az `Osztalyom.f` függvény létezik (`x` objektum az `Osztalyom` példánya), de `x.i` nem érvényes ha `Osztalyom.i` változót nem hoztuk létre az osztály definiálásakor. Fontos, hogy `x.f` nem ugyanaz, mint `Osztalyom.f` — ez egy *Objektum metódus, nem egy függvény objektum.* Fordító megjegyzése: az osztálydefiníció egy minta, ami alapján a konkrét objektumokat létrehozza a Python. Van egy prototípus, ami alapján legyártható több példány - a prototípus nyilván nem ugyanaz, mint a róla mintázott újabb egyedek. Más nyelvekben jártas programozóknak zavaró lehet hogy a tutorial a függvényeket ugyanúgy jelöli, mint a változókat: `x.f` egy függvény, míg például a php-ban ezt a tényt az üres zárójelekkel külön jelölik: `x.f()`

9.3.4. Az objektum metódusok

Többnyire a metódusokat közvetlenül használjuk:

```
x.f()
```

//:: // 2004. okt. 9:

Példánkban `x.f` a `'hello vilag'` string-el tér vissza. Ezt a függfényt nem csak közvetlenül hívhatjuk meg: ?? `x.f` egy objektum metódus, tárolható és később is hívható, például így:

```
xf = x.f
while True:
    print xf()
```

Ez a kód az örökkévalóságig a ‘`hello vilag`’ üzenetet írja ki.

Pontosan mi történik egy objektummetódus hívásakor? Lehet, hogy már észrevetted hogy a `x.f()`-t a fenti példában paraméterek nélkül hívtuk meg - annak ellenére, hogy `f` függvénydefiníciója egy paraméter használatát

előírja. Mi van ezzel a paraméterrel? Szerencsére a Pythonban ha egy paramétert igénylő függvényt paraméter nélkül próbálunk használni, kivételdobás történik.

Lehet hogy már kitaláltad a választ: az a különleges a metódusokban, hogy hívásukkor az őket tartalmazó osztálypéldányt megkapják az első változóban. A példánkban `x.f()` hívása pontosan ugyanaz, mintha `Osztalyom.f(x)` metódust hívnánk. Ford. megjegyzése: valójában a függvénynek itt is jelezzük, hogy az `x` egyedről van szó, a paraméter miatt. Általában metódusok hívása *n* paraméterrel ugyanaz, mintha az osztálydefiníció függvényét hívnánk meg úgy, hogy a legelső paraméter elé az aktuális példány nevét beillesztjük.

Ha nem értenél valamit a metódusok működéséről, nézz meg kérlek néhány gyakorlati példát. Amikor egy példány tulajdonságára hivatkozol, és az nem létezik a változók között, az értelmező az osztálydefinícióban fogja keresni. Ha a név egy érvényes osztálytulajdonságra mutat, ami egy fügvény, a fenti példában szereplő folyamat történik: az értelmező az `x.f()` hívást átalakítja – a paramétereket kigyűjti, majd első paraméterként `x`-et tartalmazva létrehoz egy új paraméterlistát és meghívja a `Osztalyom.f(x, paraméter1, p2...)` függvényt. Az adott példányban tehát a függvény valójában nem is létezik (absztrakt objektum).

Az adat attribútumok felülírják az ugyanolyan nevű metódusokat; a névütközések elkerülése végett (amelyek nagyon nehezen megtalálható programhibákhoz vezethetnek) érdemes betartani néhány elnevezési szabályt, melyekkel minimalizálható az ütközések esélye. Ezek a szabályok például a metódusok nagybetűvel írását, az adat attribútumok kisbetűs írását – vagy alsóvonás karakterrel történő írását jelentik; vagy igék használatát a metódusokhoz, és főnevekét az adat attribútumokhoz.

Az adat attribútumokra a metódusok is hivatkozhatnak, éppúgy mint az objektum hagyományos felhasználói. Más szavakkal az osztályok nem használhatók csupasz absztrakt adattípusok megvalósítására. Valójában a Pythonban jelenleg semmi sincs, ami az adatrejtés elvét biztosítani tudná – minden az elnevezési konvenciókra épül. Másrészről az eredeti C alapú Python képes teljesen elrejteni a megvalósítási részleteket és ellenőrizni az objetkum elérését, ha szükséges; ehhez egy C nyelven írt kiegészítést kell használni.

A kliensek az adat-atribútumokat csak óvatosan használhatják, mert elronthatják azokat a variánsokat, amelyeket olyan eljárások tartanak karban, amelyek időpont-pecsételéssel dolgoznak. Az objektum felhasználói saját adat attribútumaikat bármiféle ellenőrzés nélkül hozzáadhatják az objektumokhoz amíg ezzel nem okoznak névütközést — az elnevezési konvenciók használatával elég sok fejfájástól megszabadulhatunk!

A metódusokon belül nem létezik rövidítés, gyors hivatkozás az adat attribútumokra. Én úgy látom, hogy ez növeli a metódusok olvashatóságát, és nem hagy esélyt a helyi és a példányosított változók összekeverésére, mikor a metódus forráskódját olvassuk.

A hagyományokhoz hűen a metódusok első paraméterének neve rendszerint `self`. Ez valóban csak egy szokás: a `self` névnek semmilyen speciális jelentése nincs a Pythonban. (Azért vegyük figyelembe, hogy ha eltérünk a hagyományoktól, akkor a program nehezebben olvashatóvá válik, és a *class browser - osztály böngésző* is a tradicionális változónevet használja.

Az osztály definíciójában megadott függvények az osztály példányai számára hoznak létre metódusokat (a példányhoz tartozó függvényeket). Nem szükségszerű azonban hogy egy függvénydefiníció kódja az osztálydefiníció része legyen: egy definíción kívüli függvény helyi változóhoz való rendelése is megfelel a célnak. Például:

```
% # Function defined outside the class
# Egy osztályon kívül definiált függvény
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello vilag'
    h = g
```

Most `f`, `g` és `h` egyaránt `C` osztály attribútumai (gyakorlatilag objektum hivatkozások) — következésképpen `C` osztály minden példányának metódusai is— `h` és `g` pedig valójában ugyanazt a függvényt jelentik. Azért ne feledjük, hogy a fenti példa használata a program olvasóját összekavarhatja!

Az osztályon belüli metódusok egymást is hívhatják a `self` argumentum használatával:

```
class Taska:
    def __init__(self):
        self.tartalom = []
    def belerak(self, x):
        self.tartalom.append(x)
    def belerak_ketszer(self, x):
        self.belerak(x)
        self.belerak(x)
```

A metódusok a globális névtérben lévő függvényekre is hasonlóképp hivatkozhatnak. (Maguk az osztálydefiníciók soha nem részei a globális névtérnek!) Míg egy kivételes esetben a globális névtér változóinak használata jól jöhet, több esetben is jól jöhet a globális névtér elérése: a globális névtérbe importált függvényeket és modulokat az adott osztálymetódusból is használhatjuk, mintha az adott függvényben vagy osztályban definiálták volna őket. Rendszerint az osztály az önmaga által definiált metódust a globális névtérben tartja, és a következő részben meglátjuk majd, miért jó ha a metódusok a saját osztályukra hivatkozhatnak!

9.4. Öröklés

Természetesen az öröklés támogatása nélkül nem sok értelme lenne az osztályok használatának. A származtatott osztályok definíciója a következőképpen néz ki:

```
class SzarmaztatottOsztalyNeve(SzuloOsztalyNeve):
    <utasitas-1>
    .
    .
    .
    <utasitas-N>
```

a `SzuloOsztalyNeve` névnek a származtatott osztály névterében léteznie kell. Abban az esetben, ha a szülőosztály másik modulban lett definiálva, a modulnev.szuloosztalynev formát is használhatjuk:

```
% class DerivedClassName(modname.BaseClassName):
class SzarmaztatottOsztalyNeve(modulnev.SzuloOsztalyNeve):
```

A származtatott osztály definíciójának feldolgozása hasonló a szülőosztályokéhoz — az osztályobjektum létrehozásakor a szülőosztály is a példány része lesz. Ha egy osztály-attribútumra hivatkozunk, és az nincs jelen az osztályban, az értelmező a szülőosztályban keresi azt — és ha az szintén származtatott osztály, akkor annak a szülőjében folytatódik a keresés, rekurzívan.

A származtatott osztályok példányosításában nincs semmi különleges: `SzarmaztatottOsztalyNeve()` létrehozza az osztály új példányát. A metódusok nyilvántartását a következőképp oldja meg az értelmező: először az aktuális osztály megfelelő nevű osztály-objektumait vizsgálja meg, majd ha nem találja a keresett metódust, elindul a szülőosztályok láncán. Ha a keresési folyamat eredményes, tehát valamelyik szülőosztályban megtalálta a keresett nevű objektumot, az adott objektumnév hivatkozása érvényes lesz: a talát objektumra mutat.

A származtatott osztályok felülírhatják a szülőosztályok metódusait. A metódusoknak nincsenek különleges jogaik más, ugyanabban az objektumban lévő metódusok hívásakor. A szülőosztály egy metódusa `/a_eredeti()/`, amely ugyanazon szülőosztály egy másik metódusát hívja `/b()`, lehet hogy nem fog működni, ha a származtatott osztályból felülírják `/a_uj()/, ami nem hívja már b()-t`. (C++ programozóknak: a Pythonban minden metódus valójában `virtual` típusú.

A származtatott osztály metódusa, amely felülírja a szülőosztály egy metódusát, valójában inkább kiterjeszti az eredeti metódust, és nem egyszerűen csak kicseréli. A szülőosztály metódusára így hivatkozhatunk: '`SzuloOsztalyNev.metodusnev(self, paraméterek)`'. Ez néha jól jöhet. (Fontos, hogy ez csak akkor működik, ha a szülőosztáy a globális névtérben lett létrehozva, vagy közvetlenül beimportálva.)

9.4.1. Többszörös öröklés

A python korlátozottan támogatja a többszörös öröklést is. Egy ilyen osztály definíciója a következőképp néz ki:

```
% class DerivedClassName(Base1, Base2, Base3):
%     <statement-1>
class SzarmaztatottOsztalyNeve(Szulo1, Szulo2, Szulo3):
      <utasitas-1>

    .
    .
    .
%     <statement-N>
      <utasitas-N>
```

Az egyedüli nyelvtani szabály amit ismerni kell, az osztály attribútumok feloldásának a szabálya. Az értelmező először a mélyebb rétegekben keres, balról jobbra. A fenti példában ha az attribútum nem található meg a `SzarmaztatottOsztaly`-ban, akkor először a `Szulo1`-ben keresi azt, majd rekurzívan a `Szulo2`-ben, és ha ott nem találja, akkor lép tovább rekurzívan a többi szülőosztály felé.

Néhányan első pillanatban arra gondolnak, hogy a `Szulo2` -ben és a `Szulo3`-ban kellene előbb keresni, a `Szulo1` előtt — mondván hogy ez természetesebb lenne. Ez az elgondolás viszont igényli annak ismeretét, hogy mely attribútumot definiáltak a `Szulo1`-ben vagy annak egyik szülőosztályában, és csak ezután tudod elkerülni a `Szulo2`-ben lévő névütközéseket. A mélyebb először szabály nem tesz különbséget a helyben definiált és öröklött változók között.

Ezekből gondolom már látszik, hogy az átgondolatlanul használt többszörös öröklődés a program karbantartását rémálommá teheti – a névütközések elkerülése végett pedig a Python csak a konvenciókra támaszkodhat. A többszörös öröklés egyik jól ismert problémája ha a gyermekosztály két szülőosztályának egy közös nagyszülő osztálya van. Ugyan egyszerű kitalálni hogy mi történik ebben az esetben (a nagyszülő adat attribútumainak egypéldányos változatát használja a gyermek) – az még nem tisztázott, hogy ez a nyelvi kifejezésmód minden esetben használható-e.

9.5. Private Variables

9.6. Egyéni változók

Egyedi azonosítók létrehozását az osztályokhoz a Python korlátozottan támogatja. Bármely azonosító, amely így néz ki: `__spam` (legalább két bevezető alsóvonás, amit legfeljebb egy alsóvonás követhet) ??? ez bizonytalan szövegesen kicserélődik a `_classname__spam` formára, ahol a `classname` az aktuális osztály neve egy alsóvonással bevezetve. Ez a csere végrehajtódik az azonosító pozíciójára való tekintet nélkül, úgyhogy használható osztály-egyedi példányok, osztályváltozók, metódusok definiálására — még akkor is, ha más osztályok páldányait saját privát változói közé veszi fel (Ford: ellenőrizni!)

Ha a cserélt név hosszabb mint 255 karakter, az értelmező csonkíthatja az új nevet. Külső osztályoknál, vagy ahol az osztálynév következetesen alsóvonásokból áll??? nem történik csonkítás.

A névcsonkítás célja az, hogy az osztályoknak egyszerű megoldást biztosítson a „private” változók és metódusok definiálására — anélkül, hogy aggódnunk kellene a származtatott osztályokban megjelenő privát változók miatt, vagy esetleg a származtatott osztályban már meglévő változó privát változóval való felülírása miatt. Fontos tudni, hogy a névcsonkítási szabályok elsősorban a problémák elkerülését célozzák meg — így még mindig leheséges annak, aki nagyon akarja, hogy elérje vagy módosítsa a privátnak tartott változókat.

Ez speciális körülmények között nagyon hasznos lehet, például hibakeresésnél, és ez az egyik oka annak, hogy ezt a kibúvót még nem szüntették meg.

(Buglet (duda): származtatás egy osztályból a szülőosztály nevével megegyező néven – a szülőosztály privát változóinak használata ekkor lehetséges lesz.)

Fontos, hogy a `exec`, `eval()` vagy `evalfile()` által végrehajtott kód nem veszi figyelembe a hívó osztály nevét az aktuális osztály esetében — ez hasonló a `global` változók működéséhez, előre lefordított byte-kód esetében. Hasonló korlátozások léteznek a `getattr()`, `setattr()` és `delattr()` esetében, ha közvetlenül hívják meg a `__dict__` utasítást.

9.7. Egyebek...

Alkalomadtán hasznos lehet a Pascal „record”, vagy a C „struct” adattípusaihoz hasonló szerkezetek használata – egybefogni néhány összetartozó adatot. A következő üres osztálydefiníció ezt szépen megvalósítja:

```
% class Employee:
class Alkalmazott:
    pass

% john = Employee() # Create an empty employee record
john = Alkalmazott() # Egy üres alkalmazott rekordot hoz létre

% # Fill the fields of the record
% john.name = 'John Doe'
% john.dept = 'computer lab'
% john.salary = 1000
# A rekord mezőinek feltöltése
john.nev = 'John Doe'
john.osztaly = 'számítógépes labor'
john.fizetes = 1000
```

Ez a kis kódrészlet ami egyéni adat típusokat kezel, gyakran követ adatszerkezetek tárolására való osztálydefiníciókat.

Az egyes objektumpéldányok saját tulajdonságokkal is rendelkeznek: `m.im_self` az aktuális objektumpéldányra mutat ??? ezt ki kell próbálni???, `m.im_func` az objektumban elérhető metódusokat tartalmazza. ??? ezt is próbáld ki

9.8. Kivételek alkalmazása az osztályokban

A felhasználói kivételek az osztályokban is működnek — használatukkal egy bővíthető, hierarchikus kivételstruktúrát építhetünk fel.

A raise utasításnak kétféle használati módja lehetséges:

```
raise Osztaly, peldany

raise peldany
```

Az első esetben `peldany`-nak az Osztaly-ból kell származnia. A második eset egy rövidítés:

```
% raise instance.__class__, instance
raise peldany.__class__, peldany
```

Az except záradékban lévő class megegyezik egy kifejezéssel ha az ugyanaz az osztály vagy a szülőosztály (de nem másik kerülő úton — az except záradékban lévő származtatott osztály figyelése nem egyezik meg a szülőosztály figyelésével.) Például a következő kód B, C, D kimenetet fog produkálni:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with ‘`except B`’ first), it would have printed B, B, B — the first matching except clause is triggered.

Fontos, hogy ha az except záradékot megfordítjuk (‘`except B`’ van először), B, B, B lesz a kimenet — az első except-re való illeszkedés után a többit már nem vizsgálja az értelmező.

Mikor egy kezeletlen kivétel miatt hibaüzenetet küld az értelmező, és a hiba egy osztályból származik, a kimenetben szerepel az osztály neve, egy kettőspont, egy space, és befejezésül a példányt stringgé konvertálja az értelmező a beépített `str()` függvénnyel.

9.9. Bejárók

Valószínűleg már észrevetted, hogy a legtöbb tároló objektum bejárható a `for` ciklusutasítás használatával:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

A ciklus használatának módja tiszta, tömör és kényelmes. A ciklusok a Python egyik nyelvi alappillérét alkotják. A színfalak mögött a `for` utasítás meghívja a `iter()` függvényt, amely a bejárandó objektumhoz tartozik. Ez a függvény egy ciklus objektummal tér vissza, ami meghatározza a `next()` metódust, amellyel az objektum elemeit lehet elérni. Ha a függvény eléri az objektum utolsó elemét, és a következő elemre akarunk lépni, `StopIteration` kivételt okozunk, amely a `for` ciklust megszakítja. És lássuk mindezt a gyakorlatban:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration
```

A ciklusok működésébe bepillantva már könnyű saját bejáró eseményt készíteni egy osztályhoz. Definiálni kell az `__iter__()` metódust, ami a `next()` függvény eredményeképp létrejövő objektummal tér vissza. Ha az osztály definiálja a `next()` metódust, akkor az `__iter__()` egyszerűen a `self` objektummal tér vissza:

```
>>> class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
print char

m
a
p
s
```

9.10. Generátorok

A generátorok egyszerű és hatékony eszközök (adat)bejárók készítésére. A normális függvényekhez hasonló a felépítésük, de használják a `yield` utasítást ha adatot szeretnének visszaadni. A `next()` metódus minden hívásakor a generátor ott folytatja az adatok feldolgozását, ahol az előző hívásakor befejezte (emlékszik minden változó értékére, és hogy melyik utasítást hajtotta végre utoljára). Lássunk egy példát arra, hogy milyen egyszerű egy generátort készíteni:

```
>>> def reverse(data):
        for index in range(len(data)-1, -1, -1):
            yield data[index]

>>> for char in reverse('golf'):
        print char

f
l
o
g
```

Bármi, amit egy generátorral megtehetsz, osztály alapú bejárókkal is kivitelezhető – az előző részben leírtak szerint. Ami a generátorokat teljessé??? teszi, hogy a `__iter__()` és a `next()` metódusok automatikusan létrejönnek.

Egy másik fontos lehetőség hogy a helyi változók, és a végrehajtás állapota automatikusan tárolódik a generátor két futása között.

Az automatikus metódus generálás és program állapot mentés gondolatmenetét folytatva, amikor a generátor futása megszakad, ez az esemény automatikusan `StopIteration` kivételt vált ki. Egymással kombinálva ezek a nyelvi szolgáltatások egyszerűvé teszik a bejárók készítését néhány függvény megírásának megfelelő – viszonylag kis erőfeszítéssel.

10. fejezet

A Python alap-könyvtár rövid bemutatása - Standard Library 1.

10.1. Felület az operációs rendszerhez

Az `os` modul nagyon sok függvényt tartalmaz, melyek az operációs rendszerrel kommunikálnak:

```
>>> import os
>>> os.system('time 0:02')   # FIGYELEM: átállítottuk a rendszer óráját!
0
>>> os.system('time')
A pontos idő: 0:02:08,14
Írja be az új időt:          # ezt magyar XP-n próbáltam ki. új időt adhatunk meg.



>>> os.getcwd()       # Az aktuális könyvtár nevét adja vissza.
'C:\\Python24'
>>> os.chdir('/server/accesslogs')      # és itt könyvtárar váltottunk.
```

Fontos, hogy az importálás során az '`import os`' alakot használd, és ne a '`from os import *`' alakot. Ez megóv attól, hogy az `os.open()` függvény elfedje (és használhatatlanná tegye) a beépített `open()` függvényt, ami teljesen másképp működik

A beépített `dir()` és `help()` függvények sokat segíthetnek ha olyan nagy modulokkal van dolgod, mint például az `os`:

```
>>> import os
>>> dir(os)
<kilistázza az összes függvényt, ami az 'os' modulban található>
>>> help(os)
<egy nagyon részletes manual oldalt generál a modulok dokumentációs karakterláncából>
```

A mindennapi fájl- és könyvtár-műveletekhez az `shutil` modul magasszintű, könnyen használható felületet nyújt:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2. Karakterhelyettesítő jelek – dzsóker karakterek

A `glob` modulban lévő függvény segít a fájl listák elkészítésében, ha dzsóker karaktert használsz:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. Parancssori paraméterek

A programoknak gyakran fel kell dolgozniuk a parancssori paramétereiket. Ezek a paraméterek a `sys` modul *argv* attribútumában tárolódnak, listaként. Például ha kiadjuk azt a parancsot, hogy '`python demo.py egy ketto harom`', az a következő kimenetet eredményezi:

```
>>> import sys
>>> print sys.argv
['demo.py', 'egy', 'ketto', 'harom'] # a nulladik argumentum mindig a program neve!
```

A `getopt` modul képes feldolgozni a *sys.argv* elemeit a UNIX `getopt()` függvényének szabályai szerint. Ennél még hatékonyabb és rugalmasabb program-paraméter feldolgozást tesz lehetővé az `optparse` modul.

10.4. Hiba-kimenet átirányítása, programfutás megszakítása

The `sys` module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

A `sys` modul szintén rendelkezik *stdin*, *stdout*, és *stderr* attribútummal. Ez utóbbi használatos figyelmeztetések és hibaüzenetek láthatóvá tételére – például akkor, amikor a *stdout* át van irányítva, mondjuk egy fájlba:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

A legrövidebb út egy program megszakítására a '`sys.exit()`' utasítás..

10.5. Reguláris kifejezések - karakterláncok

A `re` modul segítségével reguláris kifejezéseket használhatsz szövegfeldolgozásra. Összetett illeszkedési és módosító szabályokat határozhatsz meg – a reguláris kifejezések rövid, tömör megoldást kínálnak:

```
>>> import re               # kovetkezik: minden f-el kezdodo szot kigyujtunk:
>>> re.findall(r'\bf[a-z]*', 'aki felveszi, az felmelegszik, aki nem, az fazik')
['felveszi', 'felmelegszik', 'fazik']

>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'macska a a kalapban a a hazban')
'macska a kalapban a hazban'    # a pelda nem tokeletes, 'a a a'-bol
# 'a a'-t csinal, de szemlelteteshez jo
```

Ha egyszerűbb szövegmódosítási igényed van, a string metódusokat javasoljuk, mert olvashatóak és a hibakeresés is könyebb velük.

```
>>> 'Teat Peternek'.replace('Peter', 'Elemer')
'Teat Elemernek'
```

10.6. Matematika

A `math` modulon keresztül érhetőek el a háttérben működő C függvények, melyekkel lebegőpontos műveleteket végezhetsz:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

A `random` modullal véletlenszámokat generálhatsz:

```
>>> import random
>>> random.choice(['alma', 'korte', 'banan'])
'alma'
>>> random.sample(xrange(100), 10)   # ismetles nelkuli mintavetel
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # véletlen egész szám kiválasztása 0-6ig terjedő tartományban
4
```

10.7. Internet elérés

Több modul is van, amely lehetővé teszi az Internet elérését, és különböző protokollok használatát. A két legegyszerűbb az `urllib2` – adatfogadás url címekről, és az `smtplib` modul, amellyel levelet küldhetsz:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://www.python.org/'):
                                  # for: az oldal soronkénti feldolgozása:
...     if 'Python' in line:      # keressük azokat a sorokat,
...         print line            # ahol a Python szó megtalálható
```

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
"""To: jcaesar@example.org
From: soothsayer@example.org

Szevasz! Eljutottal a tutorial vegeig!.
""")
>>> server.quit()
```

10.8. A dátumok és az idő kezelése

A `datetime` modul biztosít osztályokat a dátumok és az időpontok manipulálására – egyszerűbbeket és összetettebbeket is. A dátum- és az idő- aritmetikai műveletek támogatottak – a középpontban a kimenet formázása és módosítása áll. A modul támogatja azokat az objektumokat is, amelyek kezelni tudják az időzónákat.

```
# a dátumok könnyen létrehozhatóak és formázhatóak:
>>> from datetime import date
>>> most = date.today()
>>> most
datetime.date(2003, 12, 2)
>>> most.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# a dátumok támogatják a naptári műveleteket:
>>> szuletesnap = date(1964, 7, 31)
>>> kor = most - szuletesnap  # a most-ot az elozo peldaban hataroztuk meg!
>>> kor.days   # days = napok, itt a napok szamat jelenti
14368
```

10.9. Tömörítés - zip, gzip, tar...

Az elterjedtebb archiváló és tömörítő formátumok közvetlenül támogatottak, a következő modulokban: `zlib`, `gzip`, `bz2`, `zipfile`, and `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. Teljesítménymérés

Néhány Python programozó komoly érdeklődést mutatott a különböző probléma-megoldások teljesítményének összehasonlítása iránt. A Pythonban található egy mérőeszköz, amely azonnali választ ad ezekre a kérdésekre.

Például használhatunk tuple becsomagolást és kicsomagolást a megszokott paraméter-átadás helyett. A `timeit` modul gyorsan demonstrál egy egyszerű teljesítmény mérést:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

A `timeit` modul apró kódrészletek végrehajtási idejének mérésére szolgál. Ezzel ellentétben a `profile` és a `pstats` modulok nagyobb kódrészletek futási-idő kritikus részeinek meghatározására szolgál.

10.11. Minőségellenőrzés

A jóminőségű programok fejlesztésnek egyik elmélete az, hogy minden függvényhez próbaadatokat, teszteket írunk – majd a fejlesztési folyamat során ezeket gyakran lefuttatjuk - így azonnal kiderül, ha a várttól eltérően viselkedik a program.

A The `doctest` modul tartalmaz olyan eszközöket, amelyekkel modulokat vizsgálhatunk, és a program dokumentációs karakterláncába ágyazott teszteket futtathatunk le. A teszt létrehozása olyan egyszerű, mint kivágni és beilleszteni egy tipikus függvényhívás során bejövő-keletkező adatokat.

Ez a lehetőség elősegíti a jobb dokumentáltságot, hiszen a felhasználónak rögtön függvényhívási példát mutathatunk – továbbá ellenőrizhetővé teszi a doctest modulnak, hogy a kód a dokumentációval összhangban van-e.

```
def atlag(ertekek):
    """Listában átadott számok számtani közepét határozza meg a függvény.

    >>> print atlag([20, 30, 70])
    40.0
    """
    return sum(ertekek, 0.0) / len(ertekek)

import doctest
doctest.testmod()   # a beágyazott tesztet automatikusan kipróbálja.
```

A `unittest` modul kicsit bonyolultabb, mint a `doctest` modul – viszont több átfogó tesztkészlet kezeléséről gondoskodik, egy különálló fájlban:

(Ehhez az előző példa lefuttatása is szükséges - hogy létezzen az average függvény!)

```
import unittest

class StatisztikaiFuggvenyekTesztelese(unittest.TestCase):

    def atlag_tesztelese(self):
        self.assertEqual(atlag([20, 30, 70]), 40.0)
        self.assertEqual(round(atlag([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, atlag, [])
        self.assertRaises(TypeError, atlag, 20, 30, 70)

unittest.main() # A parancssorból történő hívás lefuttatja a teszteket.
```

10.12. Elemekkel együtt...

A Python filozófiája: „elemekkel együtt”. A legjobban ez úgy látszik, ha észrevesszük nagyszámú moduljainak - csomagjainak kifinomultságát, összetettségét.

Például:

- Az `xmlrpclib` és a `SimpleXMLRPCServer` modulok a távoli eljáráshívásokat egyszerű műveletté teszik számunkra. A neveik ellenére nincs közvetlen XML tudásra szükség.
- Az `email` csomag egy könyvtár az elektronikus levelek kezelésére – beleértve a MIME és más RFC 2822-alapú üzeneteket is. Eltérően az `smtplib` és `poplib` moduloktól, melyek azonnali levélküldést és fogadást valósítanak meg, az email csomag teljes eszközkészlettel rendelkezik összetett üzenet-struktúrák felépítéséhez és dekódolásához – a csatolt állományokat is beleértve. Továbbá tartalmazza az Interneten használt kódoló és fejléc protokollokat.
- Az `xml.dom` és az `xml.sax` csomagok nagyon jól használhatók az elterjedt adat-cserélő formátumok

kezelésére, értelmezésére és feldolgozására Ugyanúgy a `csv` modul támogatja a csv formátum közvetlen írását és olvasását. Mindent egybevéve ezek a modulok és csomagok remekül leegyszerűsíti a Python programok és más alkalmazások közötti adatcserét.

- A kultúrális tulajdonságok beállíthatók és támogatottak számos modulban, például: `gettext`, `locale`, és a `codecs` csomagban is.

11. fejezet

Az alap-könyvtár bemutatása 2. rész

Ebben a részben néhány olyan modult vizsgálunk meg, amire a professzionális programozás során szükség lesz. Ezen modulok kisebb szkriptekben ritkán fordulnak elő.

11.1. A kimenet formázása

A `repr` modulban található `repr()` függvény lehetővé teszi a nagyméretű, mélyen egymásba ágyazott adatszerkezetek rövid, áttekinthető kijelzését:

```
>>> import repr
>>> repr.repr(set('elkelkaposztastalanitottatok'))
"set(['a', 'e', 'i', 'k', 'l', 'n', ...])"
```

A `pprint` modullal finoman szabályozhatod beépített és a felhasználó által definiált objektumok megjelenítését – úgy, hogy az az értelmező számára továbbra is feldolgozható marad. Amikor az eredmény nem fér el egy sorban, egy „csinos nyomtató” sortöréseket és behúzásokat ad a kimenethez, hogy az jól olvasható legyen:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

A `textwrap` modullal szövegblokkokak jeleníthetünk meg adott szélességű blokkban:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

A `locale` modul a különböző kultúrákhoz kötődő egyedi adatformázásokhoz fér hozzá – a locale format funkciójának grouping (csoportosítás, a következő példában helyiérték) tulajdonsága közvetlenül biztosítja az adott kultúrának megfelelő szám-kijelzést:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format("%s%.*f", (conv['currency_symbol'],
...       conv['int_frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. Szöveg-sablonok

A `string` modulban található egy nagyon hasznos osztály, a `Template`. Ez lehetőséget ad a végfelhasználóknak sablon-szövegek szerkesztésére.

A szövegbe adatmezőket a '`$`' jellel helyezhetünk el, melyek mellé kapcsos zárójelbe Python változóneveket kell írni (ez számot, betűt és alsóvonás karaktert tartalmazhat). A kapcsos zárójelpárra akkor van szükség, ha nem önálló szó a beillesztett adat, például lent a 'Peternek' szó lesz ilyen - egyébként a zárójelpár elhagyható. Egyszerű '`$`' jelet így írhatsz: '`$$`'

```
>>> from string import Template
>>> t = Template('${nev}nek kuldunk  10$$-t, hogy $cselekedet.')
>>> t.substitute(nev='Peter', cselekedet='csizmat vegyen')
'Peternek kuldunk 10$-t, hogy csizmat vegyen'
```

A `substitute` medódus `KeyError` kivételd dob, ha az adatmezőkben megadott változónevet a paraméterként átadott szótárban, vagy kulcsszavas paraméterekben nem találja. Elképzelhető olyan helyzet, hogy valamelyik változónév (vagy kulcs, ha szótárról van szó) hiányzik - ilyenkor a `safe_substitute` metódust érdemes használni, ami a hiányzó adatoknál az adatmezőt változatlanul hagyja:

```
   # ez a pelda nem fut le, hianyzik $owner
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)    # ez a pelda lefut, pedig hianyzik $owner:browse confirm saveas

'Return the unladen swallow to $owner.'
```

A Template alosztály egyedi határolójelet is tud használni. Például egy tömeges fájl-átnevező funkció esetében (pl. fénykép-karbantartó programnál) az adatmezők jelzésére használhatod a százalék jelet is, az aktuális dátum, a kép sorszáma, vagy a fájl formátumának jelzése esetén:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Add meg az atnevezes modjat: (%d-datum %n-fajl_sorszama %f-fileformatum
Add meg az atnevezes modjat: (%d-datum %n-fajl_sorszama %f-fileformatum):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '%s --> %s' % (filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. The makes it possible to substitute custom templates for XML files, plain text reports, and HMTL web reports.

A szövegsablonok másik felhasználási lehetősége a többféle kimeneti formátumot támogató programokban van. Itt a vezérlési logika a kimenettől el van választva – A kimeneti fájl felépítése más XML fájloknál, szövegfájloknál, vagy html kimenet esetében, viszont az adattartalom értelemszerűen megegyezik.

11.3. Bináris adatblokkok használata

A `struct` modulban található a `pack()` és az `unpack()` függvények, melyekkel változó hosszúságú bináris adatblokkokat kezelhetsz. A következő példa bemutatja a ZIP fájlok fejléc információinak feldolgozását (a `"H"` és az `"L"` kódcsomag jelképezi a két és négybájtos előjel nélküli számokat):

```
import struct

data = open('fajlom.zip', 'rb').read()
start = 0
for i in range(3):                      # megnezzuk az elso harom fajl fejlecet
    start += 14
    fields = struct.unpack('LLLHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size =  fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size     # tovabblepes a kovetkezo fejlechez
```

11.4. Többszálúság

A szálkezeléssel lehet az egymástól sorrendileg nem függő folyamatokat párhuzamossá tenni. A szálakkal egyidőben fogadhatjuk a program felhasználójának utasításait, miközben a háttérben a program egy feladaton dolgozik - a két folyamat (kommunikáció, és háttérben munka) egymással párhuzamosan fut.

A következő kód bemutatja a `threading` modul magasszintű használatát, ami egy külön háttérfolyamatot indít, miközben a program fut tovább:

```
import threading, zipfile

class AsyncZip(threading.Thread):
        def __init__(self, infile, outfile):
            threading.Thread.__init__(self)
            self.infile = infile
            self.outfile = outfile
        def run(self):
            f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
            f.write(self.infile)
            f.close()
            print 'Kesz a zip tomoritese ennek a fajlnak: ', self.infile

background = AsyncZip('valami.txt', 'myarchive.zip')
background.start()
print 'A foprogram tovabb fut az eloterben.'

background.join()    # Varakozas a hattermuvelet befejezesere
print 'A foprogram megvarja, hogy a hattermuvelet befejezodjon.'
```

A többszálú programok egyik legfontosabb feladata azon szálak működésének koordinálása, melyek megosztott adatokon, vagy közös erőforrásokon dolgoznak. (pl. mi történik ha két szál egyidőben akar egy fájlt írni?) A Threading modul több szinkronizációs elemet tartalmaz – ilyen a zárolás, az eseménykezelés, a feltételes változók és a szemaforok.

Igaz ugyan, hogy ezek hatékony eszközök – ám előfordulhatnak kisebb tervezési hibák is, melyek nehezen megismételhetők, és nehezen kinyomozhatók. Mivel a szálak egymástól függetlenül futnak, még végrehajtásuk sorrendje sem biztos - emiatt előfordulhat, hogy a program nem ugyanúgy viselkedik, ha egymás után többször lefuttatjuk – így a hibakeresés is nehezebbé válik.

A szálak koordinálására azt javasoljuk, hogy az erőforrások elérését egy szál biztosítsa, és használd a `Queue`

modult a többi szálból érkező kérések kezelésére. Ha egy programban a szálak közötti kommunikációt a `Queue` objektumokkal biztosítod, a program koordinálásának megtervezése egyszerűbb, olvashatóbb és megbízhatóbb lesz.

11.5. Naplózás

A `logging` modul egy összetett, finoman beállítható naplózó rendszert tartalmaz. A legegyszerűbb esetben a naplózandó üzenetek fájlba, vagy a `sys.stderr` (szabványos hibacsatornára) - küldhetők:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

A fenti példa kimenete:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Alapértelmezés szerint az információs és debug üzenetek elfojtottak, és a kimenet a szabványos hiba csatornára kerül. A kimenet célja más is lehet, például email, datagram, socket vagy akár egy HTTP szerver. Az új szűrők az üzenet prioritásától függően más-más kimenetre terelhetik a naplózandó üzenetet. A prioritások: `DEBUG`, `INFO`, `WARNING`, `ERROR`, és `CRITICAL`.

A naplózó rendszer közvetlenül a Pythonból is beállítható, vagy használhatsz konfigurációs fájlt, s így a programból való kilépés nélkül megváltoztathatod a naplózás beállításait.

11.6. Gyenge hivatkozások

A memóriakezelést a Python automatikusan végzi (hivatkozásszámlálás a legtöbb objektum esetében, és szemétgyűjtés). Az utolsó objektum-hivatkozás megsemmisülése után az elfoglalt memória felszabadul.

Ez az automatizmus a legtöbb esetben jó és hasznos, de néha szükség van az objektumok követésére mindaddig, amíg használatban vannak. Érdekes, hogy éppen ez a követés az, ami a hivatkozásokat állandóvá teszi (nem szűnnek meg).

object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

A `weakref` modulban olyan eszközök vannak, amelyekkel úgy lehet nyomon követni az objektumokat, hogy a nyomkövetéssel nem hozol létre újabb hivatkozást az objektumra. Amikor az objektumot már senki nem használja, automatikusan törlődik a weakref (gyenge referencia) táblából, és a weakref objektum erről értesítést kap. A tipikus programok tároló objektumokat tartalmaznak, melyek létrehozása erőforrásigényes:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, ertek):
...             self.ertek = ertek
...     def __repr__(self):
...             return str(self.ertek)
...
>>> a = A(10)                   # hivatkozas letrehozasa
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # itt nem keletkezik hivatkozas
>>> d['primary']                # ha az objektum meg letezik, visszaadja az erteket
10
>>> del a                       # toroljuk az egyetlen hivatkozast
>>> gc.collect()                # a szemetgyujtest azonnal lefuttatjuk
0
>>> d['primary']                # ez a bejegyzes automatikusan megszunt, nem kell kulon
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in -toplevel-
    d['primary']                # ez a bejegyzes automatikusan megszunt, nem kell kulon
  File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. Listakezelő eszközök

A legtöbb adatstruktúrának szüksége van a beépített lista típusra. Néha előfordul, hogy a listák egy másfajta megvalósítására van szükség, az eredeti listáktól eltérő viselkedéssel.

Az `array` modulban található `array()` objektum hasonlít azokhoz a listákhoz, melyek csak hasonló adattípusokat tárolnak, de ezt a tárolást sokkal tömörebben végzi. A következő példában egy számokból álló tömböt láthatunk, ahol a számok mint két bájtos előjel-nélküli bináris számként tárolódnak (típuskód: `"H"`) – ellentétben a hagyományos Python int (egész szám) objektumokból álló listákkal, ahol minden bejegyzés 16 bájtot használ.

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

A `collections` modulban lévő `deque()` objektum egy – a listához hasonló típus – viszont gyorsabban tud új elemet felvenni a lista végére és elemet kiemelni a lista elejéről. Hátránya viszont, hogy a keresésben lassabb - nehézkesebb, mint a hagyományos lista. Ez az objektumtípus hasznos várakozási sorok, listák megvalósítására:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Ráadásul az Alap-Könyvtár más eszközöket is tartalmaz, például a `bisect` modult, ami rendezett listák módosítására szolgál:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

A `heapq` modulban található függvényekkel megvalósíthatók a hagyományos listákon alapuló adathalmazok kezelése. A legalacsonyabb értékű bejegyzés mindig a nulla pozícióba kerül. Ez hasznos, ha a programodnak gyakran kell elérnie a lista legkisebb elemét, de nem akarod a listát teljes mértékben rendezni:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # atrendezi a listat
>>> heappush(data, -5)                 # uj elemet ad a listaba
>>> [heappop(data) for i in range(3)]  # kilistazza a harom legkisebb elemet.
[-5, 0, 1]
```

11.8. Lebegőpontos Aritmetika

A `decimal` modulban található a `Decimal` adattípus, lebegőpontos számításokhoz. A beépített `float` típushoz képest, amely a bináris lebegőpontos számításokhoz készült, az új osztály nagyon sokat segít pénzügyi programoknál (és ott, ahol véges decimális ábrázolást használnak, a pontosság ellenőrzésével, a törvényeknek vagy a szabályoknak megfelelő kerekítés használatával, a fontos számjegyek nyomkövetésével – vagy olyan programoknál, ahol a felhasználó kézzel végzett számításokhoz akarja hasonlítani a végeredményt.

Például számítsuk ki az 5%-os adóját egy 70 centes telefonköltségnek, ami különböző eredményt ad decimális és bináris lebegőpontos számítás használata esetén. A különbség fontos lesz, ha a kerekítés a legközelebbi centhez történik:

```
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

A `Decimal` osztály eredménye egy lezáró nullát tartalmaz, automatikusan négy számjegyűen kerül ábrázolásra,

a 2*2 számjegyű (tizedesjegyek) szorzás eredményeképp. A `Decimal` ugyanolyan matematikát használ, mint amit a papíron végzett számolás, és elkerüli azokat a kérdéseket, amikor a bináris lebegőpontos számítás nem tud abszolút pontosan ábrázolni decimális mennyiségeket.

A `Decimal` osztály teljesen pontosan ábrázolja a maradékos osztást, és az egyenlőségtesztelést, ami a bináris lebegőpontos ábrázolás esetén helytelen eredményre vezet:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

A `decimal` modulban a számítások pontosságát szükség szerint beállíthatod:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
```

12. fejezet

What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Now what should you do?

You should read, or at least page through, the *Python Library Reference*, which gives complete (though terse) reference material about types, functions, and modules that can save you a lot of time when writing Python programs. The standard Python distribution includes a *lot* of code in both C and Python; there are modules to read UNIX mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and a lot more; skimming through the Library Reference will give you an idea of what's available.

The major Python Web site is http://www.python.org/; it contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location. A more informal site is http://starship.python.net/, which contains a bunch of Python-related personal home pages; many people have downloadable software there. Many more user-created Python modules can be found in the Python Package Index (PyPI).

For Python-related questions and problem reports, you can post to the newsgroup comp.lang.python, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ), or look for it in the 'Misc/' directory of the Python source distribution. Mailing list archives are available at http://www.python.org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

A. Függelék

Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the *GNU Readline* library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the UNIX and CygWin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

A.1. Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: `C-A` (Control-A) moves the cursor to the beginning of the line, `C-E` to the end, `C-B` moves it one position to the left, `C-F` to the right. Backspace erases the character to the left of the cursor, `C-D` the character to its right. `C-K` kills (erases) the rest of the line to the right of the cursor, `C-Y` yanks back the last killed string. `C-underscore` undoes the last change you made; it can be repeated for cumulative effect.

A.2. History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. `C-P` moves one line up (back) in the history buffer, `C-N` moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the `Return` key passes the current line to the interpreter. `C-R` starts an incremental reverse search; `C-S` starts a forward search.

A.3. Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called '~/.inputrc'. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for `Tab` in Python is to insert a `Tab` character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your '~/.inputrc'. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using `Tab` for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the `Tab` key to the completion function, so hitting the `Tab` key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final '`.`' and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

[1] Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

A.4. Commentary

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

B. Függelék

Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

```
0.125
```

has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction

```
0.001
```

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

```
0.3
```

or, better,

```
0.33
```

or, better,

```
0.333
```

and so on. No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation. This is why you see things like:

```
>>> 0.1
0.10000000000000001
```

On most machines today, that is what you'll see if you enter 0.1 at a Python prompt. You may not, though, because the number of bits used by the hardware to store floating-point values can vary across machines, and Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

instead! The Python prompt (implicitly) uses the builtin `repr()` function to obtain a string version of everything it displays. For floats, `repr(`*float*`)` rounds the true decimal value to 17 significant digits, giving

```
0.10000000000000001
```

`repr(`*float*`)` produces 17 significant digits because it turns out that's enough (on most machines) so that `eval(repr(`*x*`)) ==` *x* exactly for all finite floats *x*, but rounding to 16 digits is not enough to make that true.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Python's builtin `str()` function produces only 12 significant digits, and you may wish to use that instead. It's unusual for `eval(str(`*x*`))` to reproduce *x*, but the output may be more pleasant to look at:

```
>>> print str(0.1)
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the *display* of the true machine value.

Other surprises follow from this one. For example, after seeing

```
>>> 0.1
0.10000000000000001
```

you may be tempted to use the `round()` function to chop it back to the single digit you expect. But that makes no difference:

```
>>> round(0.1, 1)
0.10000000000000001
```

The problem is that the binary floating-point value stored for "0.1" was already the best possible binary approximation to 1/10, so trying to round it again can't make it better: it was already as good as it gets.

Another consequence is that since 0.1 is not exactly 1/10, adding 0.1 to itself 10 times may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See *The Perils of Floating Point* for a more complete account of other common surprises.

As that says near the end, „there are no easy answers." Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the discussion of Python's `%` format operator: the `%g`, `%f` and `%e` format codes supply flexible and easy ways to round float results for display.

B.1. Representation Error

This section explains the „0.1" example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1
0.10000000000000001
```

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N* where *J* is an integer containing exactly 53 bits. Rewriting

```
 1 / 10 ~= J / (2**N)
```

as

```
J ~= 2**N / 10
```

and recalling that *J* has exactly 53 bits (is `>= 2**52` but `< 2**53`), the best value for *N* is 56:

```
>>> 2**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794L
```

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```
7205759403792794 / 72057594037927936
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be *exactly* 1/10!

So the computer never „sees” 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```
>>> .1 * 2L**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. Rounding that to 17 significant digits gives the 0.10000000000000001 that Python displays (well, will display on any 754-conforming platform that does best-possible input and output conversions in its C library — yours may not!).

C. Függelék

History and License

C.1. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2. Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.4.1

1. This LICENSE AGREEMENT is between the Python Software Foundation („PSF"), and the Individual or Organization („Licensee") accessing and otherwise using Python 2.4.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.4.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., „Copyright © 2001-2004 Python Software Foundation; All Rights Reserved" are retained in Python 2.4.1 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.4.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.4.1.

4. PSF is making Python 2.4.1 available to Licensee on an „AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.4.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.4.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.4.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.4.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com („BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization („Licensee") accessing and otherwise using this software in source or binary form and its associated documentation („the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an „AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT

LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the „BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 („CNRI"), and the Individual or Organization („Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., „Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): „Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an „AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable

material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the „ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3. Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1. Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/ matumoto/MT2002/emt19937ar.htm
The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

C.3.2. Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ‘‘AS IS’’ AND
GAI_ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR GAI_ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3. Floating point exception control

The source for the `fpectl` module includes the following notice:

C.3.4. MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
```

C.3.5. Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6. Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.7. Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) without fee is hereby granted,
provided that the above copyright notice appears in all copies, and
that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.8. Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.9. UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
```

C.3.10. XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

D. Függelék

Glossary

>>> The typical Python prompt of the interactive shell. Often seen for code examples that can be tried right away in the interpreter.

... The typical Python prompt of the interactive shell when entering code for an indented code block.

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

byte code The internal representation of a Python program in the interpreter. The byte code is also cached in the `.pyc` and `.pyo` files so that executing the same file is faster the second time (compilation from source to byte code can be saved). This „intermediate language" is said to run on a „virtual machine" that calls the subroutines corresponding to each bytecode.

classic class Any class which does not inherit from `object`. See *new-style class*.

coercion The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` builtin function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has builtin support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

descriptor Any *new-style* object that defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, writing *a.b* looks up the object *b* in the class dictionary for *a*, but if *b* is a descriptor, the defined method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

dictionary An associative array, where arbitrary keys are mapped to values. The use of `dict` much resembles that for `list`, but the keys can be any object with a `__hash__()` function, not just integers starting from zero. Called a hash in Perl.

duck-typing Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code

improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style that is common in many other languages such as C.

__future__ A pseudo module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By actually importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

generator A function that returns an iterator. It looks like a normal function except that values are returned to the caller using a `yield` statement instead of a `return` statement. Generator functions often contain one or more `for` or `while` loops that `yield` elements back to the caller. The function execution is stopped at the `yield` keyword (returning the result) and is resumed there when the next element is requested by calling the `next()` method of the returned iterator.

generator expression An expression that returns a generator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL See *global interpreter lock*.

global interpreter lock The lock used by Python threads to assure that only one thread can be run at a time. This simplifies Python by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of some parallelism on multi-processor machines. Efforts have been made in the past to create a „free-threaded" interpreter (one which locks shared data at a much finer granularity), but performance suffered in the common single-processor case.

IDLE An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment that ships with the standard distribution of Python. Good for beginners, it also serves as clear example code for those wanting to implement a moderately sophisticated, multi-platform GUI application.

immutable An object with fixed value. Immutable objects are numbers, strings or tuples (and more). Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

interactive Python has an interactive interpreter which means that you can try out things and immediately see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted Python is an interpreted language, as opposed to a compiled one. This means that the source files can be run directly without first creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

iterable A container object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the builtin function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data is available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code that attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

list comprehension A compact way to process all or a subset of elements in a sequence and return a list with the results. `result = ["0x%02x" %x for x in range(256) if x %2 == 0]` generates a list of strings containing hex numbers (0x..) that are even and in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

mapping A container object (such as `dict`) that supports arbitrary key lookups using the special method `__getitem__()`.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

mutable Mutable objects can change their value but keep their `id()`. See also *immutable*.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and builtin namespaces as well asnested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which modules implement a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules respectively.

nested scope The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class Any class that inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

Python3000 A mythical python release, not required be backward compatible, with telepathic interface.

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence An *iterable* which supports efficient element access using integer indices via the `__getitem__()` and `__len__()` special methods. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

Zen of Python Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing „`import this`" at the interactive prompt.

TÁRGYMUTATÓ

R

S

U

Z