Tutorial Python

Versiune 2.2

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

9. Decembrie 2001

PythonLabs
Email: python-docs@python.org

Abstract

Python este un limbaj de programare puternic şi uşor de învăţat. Are structuri de date de nivel înalt, eficiente şi o simplă dar eficientă abordare a programării orientate pe obiect.

Sintaxa elegantă şi natura sa interpretată, fac din Python un limbaj ideal pentru elaborarea de script-uri şi dezvoltarea rapidă de aplicaţii în multe domenii, pe majoritatea platformelor.

Interpretorul Python şi librăria standard sunt oferite sub formă de surse sau compilate pentru majoritatea platformelor existente, pe site-ul Python, http://www.python.org/, şi pot fi distribuite gratuit. Pe acelaşi site puteţi găsi distribuţii Python şi module scrise de alţi programatori, precum şi documentaţie adiţională.

Interpretorul Python este uşor de extins prin adăugarea de noi funcţii şi tipuri de date implementate în C sau C++ (sau alte limbaje apelabile din C). Python poate fi folosit şi drept limbaj de extensie pentru aplicaţii.

Acest material informează cititorul despre conceptele de bază şi capabilităţile sistemului şi limbajului Python. Ar fi de preferat să aveţi la îndemână un interpretor Python pentru a experimenta, dar toate exemplele cuprinse în acest material pot fi înţelese fără a fi imperativă prezenţa interpretorului.

Pentru o descriere a obiectelor şi modulelor standard, consultaţi documentul *Python Library Reference*. Documentul *Python Reference Manual* dă o definiţie mai exactă a limbajului. Dacă doriţi să extindeţi sau să implementaţi limbajul Python într-o aplicaţie cu ajutorul C++ consultaţi documentul *Extending and Embedding the Python Interpreter* şi *Python/C API Reference*. Există şi cărţi care acoperă limbajul Python în toată complexitatea sa.

Acest tutorial nu încearcă să acopere toate facilităţile Python-ului, sau măcar cele mai folosite dintre ele. În schimb, vă prezintă cele mai interesante şi importante facilităţi, şi vă va da o idee despre modul de funcţionare al limbajului.

După ce veţi citi acest tutorial, veţi putea citi şi scrie module sau programe Python, şi veţi fi pregătiţi să învăţaţi mai multe despre diferitele librării incluse în distribuţia Python, descrise în documentul *Python Library Reference*.

CUPRINS

CAPITOL

UNU

De ce Python?

Dacă aţi construit vreodată un shell complex, probabil cunoaşteţi senzaţia: vreţi să adăugaţi încă o facilitate, dar este deja atât de lent, atât de mare, atât de complicat; sau adăugarea facilităţii implică un apel de funcţie de sistem, sau altă funcţie care nu este accesibilă decât din C. De obicei problema pe care o aveţi de rezolvat nu este suficient de serioasă astfel încât să merite să rescrieţi shell-ul în C; poate că rezolvarea problemei necesită folosirea şir-urilor de lungime variabilă, sau a listelor sortate de nume de fişiere, lucruri accesibile într-un shell, dar care necesită multă muncă pentru a fi implementate în C; sau poate că nu sunteţi suficient de familiarizat cu limbajul C.

Să considerăm o altă situaţie: presupunem că trebuie să lucraţi cu anumite biblioteci C, şi ciclul obişnuit de scriere-compilare-testare-recompilare este prea lent. Vreţi să dezvoltaţi mult mai rapid software. E posibil să fi scris un program care poate folosi un limbaj de extensie, şi nu vreţi să dezvoltaţi un alt limbaj pentru asta, să scrieţi şi să verificaţi de erori un interpretor pentru acest limbaj, pentru ca apoi să-l adăugaţi la aplicaţia dumneavoastră.

În aceste cazuri Python este limbajul de care aveţi nevoie. Este simplu de folosit, dar este un limbaj de programare adevărat, oferind posibilitatea unei mai bune structurări, şi un suport mai bun pentru programe mari decât oferă un shell. Pe de altă parte dispune de o mult mai bună verificare a erorilor decât C-ul, şi fiind un *limbaj de nivel foarte înalt*, dispune de tipuri de date de nivel înalt cum ar fi tablouri sau dicţionare foarte flexibile, a căror implementare în C ar lua programatorului zile. Datorită caracterului general al tipurilor de date Python poate fi aplicat într-o arie mult mai largă de probleme decât AWK sau PERL. Cu toate astea, multe lucruri se realizează la fel de uşor în Python ca şi în aceste limbaje.

Python permite divizarea programelor dumneavoastră în module care pot fi folosite apoi în alte programe Python. Dispune de o colecţie vastă de module standard de la care puteţi porni în construirea programului dumneavoastră, sau pentru a învăţa programarea în Python.

Există de asemenea module încorporate care oferă facilităţi pentru sistemul de intrări – ieşiri (I/O), apeluri de sistem, sockets, şi chiar interfeţe pentru sisteme de interfaţare grafică utilizator (GUI) precum TK.

Python este un limbaj interpretat, care vă poate accelera munca în timpul procesului de dezvoltare, ne mai fiind nevoit să compilaţi codul de fiecare dată. Interpretorul poate fi folosit interactiv, ceea ce vă permite să testaţi foarte uşor anumite facilităţi ale limbajului, să scrieţi imediat programe dispensabile, sau să testaţi anumite funcţii înainte de a le folosi în programul dumneavoastră. Este foarte folositor şi drept, calculator de birou.

Python permite scrierea unor programe foarte compacte şi uşor de citit. Programele scrise în Python sunt mult mai mici decat echivalentele în C sau C++ pentru că:

- tipurile de date de nivel înalt vă permit să realizaţi operaţii complexe într-o singură instrucţiune;
- gruparea instrucţiunilor este dată de paragrafare în loc de blocuri begin/end sau de acolade;
- nu este necesară declararea variabilelor.

Python este *extensibil*: dacă aveţi cunoştinţe de C puteţi foarte uşor sa adăugaţi o nouă funcţie sau un modul, fie pentru a efectua operaţii la viteză maximă, fie pentru a lega programe Python la biblioteci disponibile numai în formă binară (biblioteci grafice). Puteţi de exemplu să adăugaţi interpretorul Python într-o aplicaţie C şi să îl folosiţi ca un limbaj de comandă sau de extensie pentru acea aplicaţie.

Numele limbajului provine de la emisiunea BBC “Monthy Python’s Flying Circus” (Circul zburător al şarpelui Monty) şi nu are nici o legătură cu disgraţioasele reptile. Referinţe la Monty Python în documentaţia ce urmează nu numai că sunt permise, sunt chiar încurajate!

1.1 În continuare

Acum că aveţi o idee despre Python, poate doriţi să îl examinaţi mai în detaliu. Din moment ce, cea mai bună metodă de a învaţa un limbaj de programare este utilizarea acestuia, sunteţi şi dumneavoastră invitaţi să o faceţi.

În urmatorul capitol este explicată utilizarea interpretorului. Informaţiile ce urmează pot părea banale, dar sunt esenţiale pentru testarea exemplelor ce vor urma.

Cealaltă parte a tutorialului prezintă diferite facilitaţi ale limbajului Python. Această prezentare este bogată în exemple, începând cu expresii simple, instrucţiuni, tipuri de date şi terminând cu funcţii, module şi concepte avansate cum ar fi tratarea excepţiilor şi clase definite de utilizator.

CAPITOL

DOI

Utilizarea interpretorului Python

2.1 Invocarea interpretorului

Interpretorul Python este deobicei instalat în '/usr/local/bin/python' pe maşinile unde există această cale. În lista cu directoarele de sistem a UNIX introduceţi calea '/usr/local/bin' pentru a putea porni interpretorul cu comanda :

```
python
```

Din moment ce directorul în care este instalat interpretorul, poate fi modificat la instalare, şi alte căi sunt posibile. Vorbiţi cu administratorul dumneavoastră de sistem înainte de a lua o decizie în privinţa directorului de instalare. (Ex: '/usr/local/python' este o alternativă foarte populară).

Tastarea unui caracter de sfârşit de fişier (`Control-D` pe UNIX, sau `Control-Z` pe DOS sau Windows) în timpul prompt-ului principal determină oprirea execuţiei interpretorului cu cod de ieşire zero. Dacă acest lucru nu funcţionează încercaţi comenzile: '`import sys; sys.exit()`'.

Facilităţile de editare în linie de comandă ale interpretorului nu sunt deloc deosebite. Pe sisteme UNIX, în timpul instalării Python, puteţi activa opţiunea de suport a bibliotecii GNU de editare în linie de comandă, care adaugă mai multe facilităţi pentru editare în linie de comandă. Pentru a vedea dacă această opţiune este activată, tastaţi combinaţia de taste `Control-P` la primul prompt Python. Dacă auziţi un sunet, aveţi activată opţiunea despre care vorbeam mai sus (consultaţi Anexa A, pentru o scurtă listă de combinaţii de taste foarte utile). Dacă nu se întamplă nimic sau este afişat ^P, opţiunea de editare a liniei de comandă nu este activată, deci nu veţi putea decât să ştergeţi (BACKSPACE) sau să introduceţi noi caractere în linie.

Interpretorul funcţionează asemanător shell-ului UNIX: Când este apelat cu intrarea standard conectată la un dispozitiv TTY, citeşte şi execută comenzi interactiv. Când este apelat cu un parametru reprezentând un nume de fişier, citeşte şi execută un *script* (program) din acel fişier.

O a treia modalitate de a porni interpretorul este '`python`' **-c** *command[arg]* ... care execută instrucţiunea/instrucţiunile din *comandă*, analog opţiunii **-c** a shell-ului. Din moment ce o comandă poate conţine spaţii sau alte caractere utilizate de shell (ca separatori de parametrii de exemplu) este bine să introduceţi comanda/comenzile între ghilimele.

Observaţi că există o diferenţă între '`python file`' şi '`python <file`'. În primul caz, datele de intrare ale programului sunt oferite de fişier. Din moment ce acest fişier a fost deja citit până la capăt înainte de începerea programului, programul va întâlni sfârşit de fişier imediat. În al doilea caz (cel mai interesant şi folosit) datele de intrare sunt oferite de orice fişier sau dispozitiv conectat la intrarea standard a interpretorului Python.

Când este rulat un script, este câteodată folositor ca după rularea script-ului să se intre în modul interactiv. Acest lucru se realizează foarte simplu folosind un parametru **-i** înainte de numele fişierului în care se află script-ul. Evident că acest parametru nu va funcţiona dacă script-ul este citit de la intrarea standard.

2.1.1 Transmiterea argumentelor

Atunci când după parametrul care reprezintă fişierul în care se află script-ul, interpretorului i se mai transmit şi alţi parametrii, acesta îi va transmite mai departe script-ului prin intermediul variabilei `sys.argv`, variabilă ce conţine o listă (vector) de şiruri de caractere. Atunci când în loc de numele fişierului se transmite `'_'` (adică intrarea standard), `sys.argv[0]` = `'_'`. Când este folosită comanda **-c**, `sys.argv[0]` = **'-c'**. Parametrii de după **-c** *comandă* nu sunt consideraţi parametrii ai interpretorului, ci sunt scrişi în `sys.argv` pentru a fi folosiţi de comandă.

2.1.2 Modul interactiv

Când comenzile sunt citite de la un `TTY`, se spune că interpretorul se află în modul *interactiv*. În acest mod există două prompt-uri. *Prompt-ul principal* (de obicei '`>>> `') şi *promptul secundar* (de obicei '`... `') folosit pentru continuarea liniilor. Interpretorul afişează imediat după pornire un mesaj de întâmpinare încare se specifică versiunea şi autorii Python-ului, urmate de primul prompter :

```
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
>>>
```

Liniile de continuare sunt folosite la introducerea unei construcţii multilinie. De exemplu:

```
>>> pamantul_este_plat = 1
>>> if pamantul_este_plat:
...     print "Ai grija sa nu cazi!"
...
Ai grija sa nu cazi!
```

2.2 Interpretorul şi mediul în care rulează

2.2.1 Tratarea erorilor

Cand are loc o eroare, interpretorul afişează un mesaj de eroare şi starea stivei în momentul respectiv. În modul interactiv revine la prompt-ul principal. Atunci când intrările provin dintr-un fişier, interpretorul se opreşte cu un cod de eroare diferit de zero după ce a afişat starea stivei.(Excepţiile tratate nu sunt erori în acest context). Unele erori sunt inevitabil fatale şi cauzează terminarea anormală cu cod de ieşire diferit de zero (Ex: lipsa de memorie). Toate mesajele de eroare sunt scrise la ieşirea pentru erori (standard error stream). Ieşirile provenite din execuţia comenzilor sunt scrise la ieşirea standard.

Dacă se apasă caracterul de întrerupere (`Control-C, DEL`) în primul prompt sau în cel secundar, comanda scrisă este pierdută revenindu-se la promptul principal. Întreruperea unei comenzi în timpul execuţiei creează o excepţie de tip `Keyboard Interrupt`, care poate fi tratată într-o instrucţiune `try`.

2.2.2 Script-uri Python Executabile

Pe sisteme UNIX,BSD, scripturile Python pot fi făcute direct executabile, la fel ca scripturile de shell, punând linia

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

(presupunem că interpretorul se află în variabila de sistem PATH a utilizatorului) la începutul script-ului şi apoi transformând fişierul în executabil. Caracterele ‘’#!’’ trebuie să fie neapărat primele două caractere din fişier.

Observaţie: Caracterul ‘#’ este folosit pentru comentarii în Python.

2.2.3 Fişierul de iniţializare al modului interactiv

Când folosiţi Python în modul interactiv, poate fi foarte folositor să aveţi câteva comenzi care să se execute automat de fiecare dată când este pornit interpretorul. Puteţi realiza acest lucru creând o variabilă de sistem cu numele PYTHONSTARTUP care să conţină numele unui fişier în care să aveţi comenzile, care doriţi să fie executate la pornirea interpretorului. Această procedură este similară, facilităţii ‘.profile’ de pe sistemele UNIX. Acest fişier de iniţializare este interpretat numai când este pornit interpretorul Python în mod interactiv. Fişierul nu este interpretat nici când interpretorul Python are ca intrare un fişier, nici când sursa explicită a comenzilor este ‘/dev/tty’. Acest fişier este executat în acelaşi cadru în care se execută şi comenzile interactive. În acest fişier pot fi schimbate chiar şi prompt-urile `sys.ps1` sau `sys.ps2`.

Dacă doriţi să executaţi şi alte fişiere de iniţializare, din directorul curent trebuie să adăugaţi câteva linii la fişierul principal de iniţializare, utilizând un cod ca acesta ‘`if os.path.isfile(’.pythonrc.py’) : execfile(’.pythonrc.py’)`’

Dacă doriţi să folosiţi acest fişier de iniţializare, într-un program, trebuie să specificaţi acest lucru explicit în cadrul programului astfel:

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

CAPITOL

TREI

O introducere scurtă în Python

În exemplele ce urmează, intrările şi ieşirile vor fi deosebite prin prezenţa sau absenţa prompt-urilor (principal '`>>> `' sau secundar '`... `'). Pentru a testa un exemplu trebuie să scrieţi linia ce apare imediat după prompt. Liniile care nu au caracterele '`>>> `' sau '`... `' în faţă sunt linii de ieşire, rezultate în urma comenzilor anterioare, sau a ultimei comenzi. Un prompt secundar, fără nimic după el, înseamnă că trebuie să introduceţi o linie goală. Acest procedeu este folosit pentru terminarea unui bloc de instrucţiuni (de tipul `'begin'...'end';{...}`).

Multe din exemplele din acest tutorial conţin comentarii. În Python comentariile încep cu un caracter '#' şi se termină la sfârşitul liniei. Un comentariu poate apărea la sfârşitul unei linii, dar nu poate apărea în cadrul unui string. Un caracter '#' în cadrul unui string nu reprezintă începutul unui comentariu ci doar un caracter '#'. Iată câteva exemple:

```
# primul comentariu
SPAM = 1                 # \c{s}i al doilea comentariu
                         # ... \c{s}i al treilea!
STRING = "# Acesta nu este un comentariu."
```

3.1 Utilizarea Python-ului drept calculator de birou

Să încercăm câteva comenzi simple Python. Porniţi interpretorul şi aşteptaţi să apară prompt-ul principal '`>>> `'.(Nu ar trebui să dureze prea mult).

3.1.1 Numere

Interpretorul funcţionează ca un simplu calculator: puteţi scrie o expresie, şi ca răspuns veţi primi rezultatul. Ordinea operaţiilor şi modul de parantezare sunt la fel ca în majoritatea limbajelor (ex: Pascal sau C).

De exemplu:

```
>>> 2+2
4
>>> # Acesta este un comentariu
... 2+2 # Un comentariu pe aceia\c{s}i linie de cod
4
>>> (50 - 5 * 6) / 4
5
>>> # Impartirea numerelor intregi are ca rezultat, de fapt,
... # partea intreaga a rezultatului impartirii
... 7 / 3
2
>>> 7 / -3
-3
```

La fel ca în C, semnul egal (‘`=`’) este folosit la atribuire. Valoarea atribuită nu constituie o ieşire (nu este tipărită pe ecran):

```
>>> latime = 20
>>> inaltime = 5*9
>>> latime * inaltime
900
```

O valoare poate fi atribuită simultan mai multor variabile:

```
>>> x = y = z = 0  # x, y \c{s}i z devin zero
>>> x
0
>>> y
0
>>> z
0
```

Există suport complet pentru virgulă mobilă. Operatorii care au operanzi de tipuri diferite vor converti operandul întreg în virgulă mobilă:

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

Numerele complexe sunt de asemenea suportate. Numerele imaginare sunt scrise cu sufixul ‘`j`’ sau ‘`J`’. Numerele complexe cu parte reală diferită de zero au forma ‘`(`*real*`+`*imag*`j)`’ sau pot fi create folosind funcţia ‘`complex(`*real*`,` *imag*`)`’:

```
>>> 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)
```

Numerele complexe sunt întotdeauna reprezentate ca două numere reale în virgulă mobilă: partea reală şi cea imaginară. Pentru a afla partea reală sau cea imaginară dintr-un număr complex *z* folosiţi `z.real` respectiv `z.imag`:

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

Funcţiile de conversie la numere reale sau întregi nu funcţionează pentru numere complexe, neexistând o metodă matematică de a transforma un număr complex într-un numar real. Folosiţi funcţia `abs(z)` pentru a calcula modulul, sau `z.real` pentru a afla partea reală:

```
>>> 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 e.g. abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

În modul interactiv ultima valoare afişată este atribuită unei variabile _. Dacă folosiţi Python ca un calculator de birou este mult mai uşor să păstraţi continuitatea calculelor:

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

Această variabilă trebuie tratată de utilizator ca având numai capacitatea de a fi citită, nu şi scrisă (READ-ONLY). Nu atribuiţi valori acestei variabile, deoarece acest lucru înseamnă că veţi creea o variabilă nouă locală care va ascunde

variabila implicită _ şi comportamentul său extraordinar.

3.1.2 Şiruri de caractere

În afară de numere Python poate manipula şi şiruri de caractere, care pot(exista) avea mai multe forme. Pot fi incluse între apostroafe sau ghilimele:

```
>>> 'omleta'
'omleta'
>>> 'n-a'
"n-a"
>>> "n-a"
"n-a"
>>> '"Da," a spus el.'
'"Da," a spus el.'
>>> "\"Da,\" a spus el."
'"Da," a spus el.'
>>> '"Nu," a spus ea.'
'"Nu," a spus ea.'
```

Şirurile de caractere pot exista pe mai multe linii, separarea realizându-se prin secvenţa de control `'\n'` indicând astfel ca linia următoare este continuarea logică a liniei curente :

```
hello = "Acesta este un \c{s}ir de caractere mai lung,\n\
care se intinde pe mai mai multe linii, exact ca \^{i}n C.\n\
   Observati ca spatiile de la inceputul liniei\
        au importanta."

print hello
```

De notat că rămâne necesar ca semnele de linie nouă să fie incluse în şir utilizând `\n`; linia nouă care urmează ultimului backslash este eliminată. Acest exemplu va afişa următoarele:

```
Acesta este un \c{s}ir de caractere mai lung,
care se intinde pe mai mai multe linii, exact ca \^{i}n C.
   Observati ca spatiile de la inceputul liniei
        au importanta.
```

Dacă construim şirul de caractere ca un şir “raw”, atunci secvenţa `\n` nu se converteşte în linie nouă, dar backslesh-ul de la sfârşitul liniei şi caracterul de linie nouă din sursă sunt incluse în şir ca date. Iată un exemplu :

```
hello = r"Acesta este un \c{s}ir de caractere mai lung,\n\
care se intinde pe mai mai multe linii, exact ca \^{i}n C."

print hello
```

va tipari:

```
Acesta este un \c{s}ir de caractere mai lung,\n\
care se intinde pe mai mai multe linii, exact ca \^{i}n C.
```

Şirurile de caractere pot exista şi între perechi de cate trei ghilimele sau apostroafe (`"""` sau `'''`). În acest caz nu aveţi nevoie de caracterele de control (ex: `\n\`) pentru a realiza separarea liniilor:

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

are următoarea ieşire:

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

Interpretorul afişează rezultatul operaţiilor cu şiruri de caractere exact în forma specificată de utilizator atunci când a introdus sau definit şirurile respectiv: între ghilimele, cu ghilimele sau alte caractere ciudate, pentru a afişa valoarea exactă a şirului. Instrucţiunea print despre care vă vom vorbi mai încolo poate fi folosită pentru a afişa şiruri fără ghilimele şi caractere de control.

Şirurile de caractere pot fi concatenate:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Două şiruri de caractere scrise unul lângă altul sunt concatenate. Prima linie din exemplul de mai sus poate fi rescrisă astfel: '`word = 'Help"A''`. Acest lucru nu merge decât pentru şiruri explicite de caractere, nu pentru orice expresie care are ca operanzi şiruri:

```
>>> import string
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> string.strip('str') + 'ing'   #  <-  This is ok
'string'
>>> string.strip('str') 'ing'     #  <-  This is invalid
  File "<stdin>", line 1, in ?
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

Şirurile de caractere sunt tratate ca vectori, la fel ca înC: primul caracter al unui şir are indicele 0. Nu există un tip de date separat pentru caracter. Un singur caracter este reprezentat ca un şir de lungime unu. Se pot specifica de asemenea subşiruri ale unui şir folosind doi indici, separaţi prin `':'`. Primul indice reprezintă poziţia de început a subşirului, iar

cel de-al doilea indice, evident indicele în şirul principal la care se termină subşirul:

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Dacă al doilea indice nu este specificat, se consideră că subşirul se termină la sfârşitul şirului.

Spre deosebire de C, în Python şirurile nu pot fi modificate. Atribuirea unei valori unui anumit caracter, sau unui subşir dintr-un şir mai mare, va genera o eroare:

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

În orice caz, creearea unui şir nou cu un conţinut combinat este suficient de simplă şi eficientă:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Omiterea primului sau celui de-al doilea indice în specificarea parametrilor unui subşir, are ca efect înlocuirea primului indice cu zero, respectiv cu lungimea şirului din care se încearcă extragerea subşirului, dacă se omite al doilea parametru.

```
>>> word[:2]    # Primele doua caractere
'He'
>>> word[2:]    # Tot \c{s}irul \^{i}nafara de primele doua caractere
'lpA'
```

Iată incă un exemplu:

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Transmiterea unor parametrii greşiţi, nu generează o eroare, pentru că în această problemă Python este foarte elegant. Astfel un indice care este prea mare va fi înlocuit cu dimensiunea şirului:

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Indicii pot fi şi numere negative, în acest caz se consideră că indicele zero se află cel mai la dreapta:

```
>>> word[-1]     # ultimul caracter
'A'
>>> word[-2]     # penultimul caracter
'p'
>>> word[-2:]    # ultimele doua caractere
'pA'
>>> word[:-2]    # totul \^{i}nafara de ultimele doua caractere
'Hel'
```

Dar, atenţie -0 este acelaşi lucru cu 0!

```
>>> word[-0]     # (deoarece -0 egal 0)
'H'
```

Indicii negativi mai mari în valoare absolută decât dimensiunea şirului sunt trunchiaţi.Acest lucru nu funcţionează şi la selectarea unui singur caracter (nu a unei subsecvenţe) :

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Mai jos aveţi un tabel care vă va ajuta la înţelegerea mecanismului de numerotare al unui şir de caractere:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Prima linie de numere reprezintă legătura indicilor pozitivi cu şirul. A doua linie reprezintă corespondenţa dintre caracterele şirului şi indicii negativi. Un subşir de la *i* la *j* este format din toate caracterele dintre cei doi indici *i*, respectiv *j*. Pentru indicii pozitivi lungimea subşirului este dată de diferenţa indicilor mărginaşi, dacă nici unul dintre ei nu este mai mare decât lungimea şirului. De exemplu lungimea subşirului `word[1:3]` este 2.

Funcţia `len()` întoarce lungimea unui şir de caractere:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.1.3 Şiruri de caractere UNICODE

Începând cu versiunea 2.0 a limbajului Python, este pus la dispoziţia programatorului un nou tip de dată, pentru manipularea textelor: obiectul Unicode. Poate fi folosit pentru stocarea şi manipularea datelor de tip Unicode(consultaţi http://www.unicode.org/) şi se integrează foarte bine cu obiectele string existente, având un mecanism de auto-conversie.

Unicode are avantajul că stabileşte o legătură unică între un număr şi orice caracter scris în scripturi moderne sau scripturi mai vechi.

La început au existat numai 256 de numere fiecare având asociat un caracter, şi toate textele erau asociate unei ”pagini de cod”(code page) care făcea legătura între caractere şi numerele asociate. acest lucru a creat foarte multă confuzie odată cu internaţionalizarea software-ului. Unicode rezolvă aceste probleme asociând o singură pagină tuturor scripturilor.

Crearea şirurilor de caractere Unicode este la fel de simplă ca şi crearea şirurilor obişniute :

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

Caracterul ‘u’ din faţa şirului indică interpretorului că trebuie să creeze un şir Unicode din expresia care urmează. Dacă doriţi să includeţi caractere speciale, o puteţi face foarte simplu folosind secvenţele Unicode-Escape. Iată un exemplu:

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

Secvenţa `\u0020` indică interpretorului să insereze caracterul cu ordinalul 0x0020 (caracterul spaţiu) înpoziţia precizată. Bineânţeles că în acest mod pot fi introduse şi alte caractere, chiar şi cele obişnuite, folosind valoarea ordinalului său ca ordinal Unicode. Datorită faptului că primele 256 de caractere Unicode sunt aceleaşi ca şi în notaţia standard Latin-1, folosită în multe ţări occidentale, procesul de introducere a caracterelor Unicode devine mult mai simplu.

Pentru experţi există şi un mod direct(raw-mode) identic ca cel fel ca pentru şirurile normale. Înaintea şirului trebuie adăugat ’ur’ pentru ca Python să intre în modul Unicode direct. Conversiile `\uXXXX` sunt utilizate numai dacă este folosit un număr impar de caractere backslash(`'\'`, înaintea caracterului `'u'`:

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

Acest mod poate fi foarte util dacă aveţi de introdus un număr mare de backslash-uri.

În afară de metodele de creare a şirurilor Unicode standard, Python oferă multe alte metode de a crea şiruri Unicode.

Funcţia built-in `unicode()` permite accesarea tuturor CODECS-urilor Unicode (CODEC = COderDECoder). Câteva dintre codificările foarte cunoscute ale caror codex-uri au fost convertite sunt : *Latin-1*, *ASCII*, *UTF-8* şi *UTF-16*. Ultimele două permit stocarea caracterelor Unicode pe unul sau mai mulţi octeţi. Codificarea implicită este ASCII, care permite treacerea caracterelor cuprinse între 0 şi 127, dar blochează celelalte caractere semnalînd eroare. Când un şir Unicode este tipărit, scris într-un fişier, sau convertit cu funcţia `str()`, conversia porneşte utilizînd codarea implicită.

```
>>> 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 ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

Pentru a converti un şir Unicode într-un şir pe 8 biţi utilizând o anumită codificare, obiectele Unicode furnizează metoda `encode()`, care are un singur argument, numele codului. Este de preferat ca numele codurilor să se scrie cu litere mici.

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

Dacă aveţi datele intr-o anume codificare şi doriţi să obţineţi un şir Unicode corespondent din ele, folosiţi funcţia `unicode()` cu numele codului ca al doilea argument.

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

3.1.4 Liste

Python pune la dispoziţia programatorului o serie de date structurate, folosite pentru a grupa, a aduna la un loc mai multe valori. Cea mai flexibilă astfel de structură este lista. O listă poate fi scrisă ca o serie de valori separate prin virgulă, şi aflată între paranteze pătrate. Elementele unei liste nu trebuie să fie neapărat de acelaşi tip:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

La fel ca şi la şiruri, primul element al unei liste are indicele 0. În acelaşi fel ca şi la şiruri, listele pot fi poziţionate, concatenate şi aşa mai departe:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

Spre deosebire se şiruri, elementele unei liste pot fi modificate:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Funcţia `len()` se aplică şi listelor :

```
>>> len(a)
4
```

Utilizarea sublistelor este de asemenea posibilă. Prin folosirea lor se poate modifica chiar şi dimensiunea listei :

```
>>> # substituirea unor componente :
... a[0:2] = [1,12]
>>> a
[1,12,123,1234]
>>> # eliminarea unor componente :
... a[0:2] = []
>>> a
[123,1234]
>>> # inserarea unor componente :
... a[1:1] = ['bletch','xyzzy']
>>> a
[123,'bletch','xyzzy',1234]
>>> # inserarea unei liste la debutul ei
... a[:0] = a
>>> a
[123,'bletch','xyzzy',1234,123,'bletch','xyzzy',1234]
>>> len(a)
8
```

Este posibil să creaţi liste din alte liste (de exemplu prin concatenare):

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Observaţi că în ultimul exemplu , `p[1]` şi `q` fac referire la acelaşi obiect. Vom reveni cu detalii despre semantica obiectelor mai târziu.

3.2 Primii paşi în programare

Bineînţeles că putem folosi Python şi la alte lucruri decât pentru a aduna 2 cu 2. Putem de exmplu să generăm o subsecvenţă din şirul lui *Fibonacci*:

```
>>> # Seria lui Fibonacci:
... # Suma a doua elemente reprezinta urmatorul element.
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Acest exemplu demonstrează următoarele noţiuni:

- Prima linie conţine o *atribuire multiplă*. Varibilele `a` şi `b` iau valorile 0 respectiv 1. Pe ultima linie, de asemenea este folosită atribuirea multiplă. Evaluarea expresiilor din partea dreaptă a unei atribuiri se face înainte de orice atribuire. Evaluarea se face de la stânga la dreapta.
- Bucla `while` se execută atâta timp cât `b < 10` (atâta timp cât condiţia este adevărată). La fel ca în C, zero înseamnă fals, şi orice număr diferit de zero înseamnă adevărat. Condiţia poate fi un şir, un element de listă, absolut orice. Orice secvenţă de lungime diferită de zero înseamnă adevărat, şi invers. Condiţia folosită în acest exemplu este o comparaţie. Comparatorii standard sunt la fel ca în C: `<`(mai mic), `>`(mai mare), `==`(egal), `<=`(mai mic sau egal), `>=`(mai mare sau egal), `!=`(diferit).
- Instrucţiunile din buclă sunt *aliniate*. Alinierea (indentation) reprezintă modul în care Python grupează instrucţiunile. Deocamdată (!) Python nu dispune de un editor în linie de comandă inteligent, astfel încât alinierea, tabularea să se facă automat. Sunteţi deci obligat să folosiţi tab sau spaţii pentru a realiza gruparea instrucţiunilor. Când veţi scrie programe Python veţi folosi un editor de text. Majoritatea editoarelor de la ora actuală realizează automat tabularea. Pentru a încheia un bloc de instucţiuni trebuie introdusă o linie goală pentru a indica editorului că editarea blocului de comenzi s-a încheiat (editorul nu poate ghici când aţi introdus ultima linie. Atenţie! fiecare instrucţiune dintr-un bloc de instrucţiuni trebuie să aibă aceleaşi număr de spaţii şi taburi înainte, deci instrucţiunile trebuie să fie **perfect** aliniate.
- Instrucţiunea `print` afişează expresia pe care o primeşte ca parametru. Şirurile de caractere sunt afişate fără ghilimele, şi un spaţiu este inserat între elemente, astfel încât afişarea diverselor valori să meargă de la sine:

  ```
  >>> i = 256*256
  >>> print 'Valoarea lui i este:', i
  Valoarea lui i este: 65536
  ```

  O virgulă la sfârşitul listei de parametrii ai instrucţiunii `print`, inhibă saltul cursorului pe linia următoare la sfârşitul instrucţiunii:

```
>>> 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
```

Observaţi că interpretorul sare la linia următoare înainte de a începe afişarea, dacă ultima linie a fost incompletă.

CAPITOL

PATRU

Structuri de control

În afară de buclele de tip `while` (explicate anterior), Python dispune de structurile de control obişnuite, întâlnite şi în celelalte limbaje.

4.1 Instrucţiuni `if`

Poate că cea mai cunoscută instrucţiune de control este instrucţiunea `if`. Exemplu:

```
>>> x = int(raw_input("Introduceti un numar \^{i}ntreg : "))
>>> if x < 0:
...     x = 0
...     print 'Negativul schimbat \^{i}nzero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Unul singur'
... else:
...     print 'Mai multi'
...
```

Pot exista mai multe secţiuni `elif` sau niciuna, iar secţiunea `else` este opţională. Cuvântul cheie '`elif`' este evident prescurtarea de la '`elseif`', şi este folositor pentru a evita tabularea excesivă. O secvenţă `if..elif..elif` funcţionează ca un bloc `case` sau `switch`, secvenţe proprii altor limbaje .

4.2 Instrucţiuni `for`

Instrucţiunea `for` din Python diferă un pic faţă de ce a-ţi întâlnit în C sau Pascal. În loc de o iteraţie dată de o progresie aritmetică (Pascal), sau de o iteraţie foarte flexibilă, pentru care programatorul poate defini atât pasul iteraţiei, cât şi condiţia de oprire (C), iteraţiile instrucţiunii Python `for` funcţionează după elementele unei secvenţe (sir sau listă).

```
>>> # Masoara marimea unor \c{s}iruri
... a = [’cat’, ’window’, ’defenestrate’]
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Nu este normal(şi sigur) ca secvenţa iterată să fie modificată în timpul secvenţei `for` (este cazul numai al secventelor modificabile, cum ar fi listele). Dacă apare necesitatea de a modifica secvenţa în timpul iteraţiei, iteraţia trebuie să fie asociată unei copii a secvenţei. Notaţia subsecventelor realizează o particularitate convenabilă :

```
>>> for x in a[:]: # copiaza intreaga lista
...     if len(x) > 6: a.insert(0, x)
...
>>> a
[’defenestrate’, ’cat’, ’window’, ’defenetrate’]
```

4.3 Funcţia `range()`

Dacă este necesară o iteraţie pe o mulţime de numere, puteţi folosi funcţia `range()` pentru a genera liste ce contin progresii aritmetice :

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

Paramentrul furnizat funcţiei `range()` nu va fi niciodată un membru al secvenţei. Este posibil ca funcţia `range()` să genereze o secvenţă începând cu un alt număr decât 0, sau raţia progresiei aritmetice poate fi modificată:

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

Pentru a realiza o iteraţie pe o mulţime de numere folosiţi funcţiile `range()` şi `len()` astfel:

```
>>> 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 Instrucţiuni `break` şi `continue`, şi clauze `else` pentru bucle

La fel ca în C, instrucţiunea `break` termină forţat orice buclă `while` sau `for`. Instrucţiunea `continue` trece necondiţionat la următoarea iteraţie. Instrucţiunile iterative pot avea şi o clauză `else`. Instrucţiunile din cadrul unei astfel de clauze `else` sunt executate atunci când bucla se termină odată cu terminarea listei (`for`) sau atunci când condiţia buclei devine falsă (pentru `while`) aceste instrucţiuni nu sunt executate dacă bucla este terminată printr-o instrucţiune `break`. Iată un exemplu care determină numerele prime până la 10:

```
>>> for n in range(2, 10):
...     for x in range (2, n):
...             if n % x == 0:
...                     print n, 'egal cu', x, '*', n/x
...                     break
...     else:
...          # bucla s-a epuizat f\u{a}ra sa g\u{a}seasc\u{a} un factor
...          print n,'este un numar prim'
...
2 este un numar prim
3 este un numar prim
4 egal cu 2 * 2
5 este un numar prim
6 egal cu 2 * 3
7 este un numar prim
8 egal cu 2 * 4
9 egal cu 3 * 3
```

4.5 Instrucţiuni `pass`

Instrucţiunea `pass` nu execută nimic. Poate fi folosită atunci când este necesară prezenţa sintactică a unei instrucţiuni, fără ca aceasta să execute ceva:

```
>>> while 1:
...     pass # Asteapta pentru intrerupere de la tastatura
...
```

4.6 Funcţii

Putem crea o funcţie care generează şirul lui Fibonacci până la o limită arbitrară:

```
>>> def fib(n): # Scrie \c{s}irul lui Fibonacci pana la n
...     "Scrie \c{s}irul lui Fibonacci pana la n"
...     a, b = 0, 1
...     while b < n:
...             print b,
...             a, b = b, a+b
...
>>> # Folosim func\c{t}ia creata
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Cuvântul cheie `def` este primul din definiţia unei funcţii. Acesteia trebuie să îi urmeze numele funcţiei şi o listă parantezată de parametrii. Instrucţiunile care formează funcţia încep pe linia imediat următoare şi trebuie să fie distanţate mai mult decât antetul funcţiei faţă de margine. Prima instrucţiune din corpul funcţiei poate fi un şir de caractere(opţional), acest şir reprezentând documentaţia funcţiei(docstring = şirul de documentaţie al funcţiei). Există utilitare care generează documentaţie pornind de la aceste şiruri, deci este o practică bună să vă folosiţi de această facilitate.

Execuţia unei funcţii crează o nouă tabelă de simboluri, folosită pentru variabilele locale. Altfel spus toate atribuirile din cadrul unei funcţii vor stoca valorile în tabela de simboluri locală. Atunci când interpretorul găseşte un nume de variabilă, întâi caută în tabela de simboluri locală, apoi în cea globală şi apoi în cea predefinită.

Cu toate acestea variabilelor globale nu li se pot atribui valori în cadrul unei funcţii, decât dacă se foloseşte instrucţiunea `global`.

Parametrii actuali ai unei funcţii apelate sunt introduşi în tabela de simboluri locală a celei funcţii înmomentul apelării ei. Astfel transferul aegumentelor se face utilizand *apel prin valoare* (unde *valoare* este totdeauna un obiect *referinţă*, nu valoarea obiectului).[1]

Când o funcţie apelează o alta funcţie, este creată o nouă tabelă de simboluri pentru funcţia apelată.

Definirea unei funcţii trece numele funcţiei în tabela de simboluri curentă.

Funcţia nou creată este recunoscută de interpretor ca o funcţie definită de utilizator. În Python există un mecanism de redenumire a funcţiilor:

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

Aţi putea obiecta că funcţia `fib` este o procedură nu o funcţie. În Python, la fel ca înC, procedurile sunt funcţii, numai că nu returnează nici o valoare. Tehnic vorbind, procedurile returnează totuşi o valoare, aceasta este `None`(niciuna). `None` este un cuvânt cheie (predefinit). În mod normal valoarea `None` nu este afişată de interpretor. Totuşi dacă doriţi să vă convingeţi că aşa funcţionează procedurile încercaţi următorul exemplu:

[1] De fapt, *apel prin referinţă* va putea fi mai bine descris, atunci daca un obiect modificabil este transmis, apelantul va vedea dacă înurma apelarii obiectul s-a modificat (un element inserat într-o listă).

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

Este foarte simplu să construim o funcţie care întoarce ca rezultat şirul lui Fibonacci:

```
>>> def fib2(n): # Intoarce \c{s}irul lui Fibonacci pana la n
...     "Intoarce o lista continand \c{s}irul lui Fibonacci pana la n"
...     result = []
...     a, b = 0, 1
...     while b< n:
...             result.append(b) # vezi mai jos
...             a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # apelarea func\c{t}iei
>>> f100             # afisarea rezultatului
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Acest exemplu demonstrează mai multe facilităţi ale limbajului Python:

- Instrucţiunea `return` întoarce rezultatul funcţiei. `return` fără nici un parametru întoarce ca rezultat al funcţiei valoarea `None`. Puteţi folosi `return` fără parametrii pentru a încheia execuţia unei funcţii înainte de a se ajunge la finalul instrucţiunilor din cadrul funcţiei. Valoarea `None` este de asemenea întoarsă ca rezultat al unei funcţii atunci când au fost executate toate instrucţiunile procedurii.
- Instrucţiunea `result.append(b)` apelează o *metodă* a obiectului `result`. O metodă este o funcţie care 'aparţine' unui obiect. Metodele sunt apelate folosind sintaxa `obj.metodă`(unde `obj` este un obiect, poate fi şi o expresie, iar `metodă` este numele metodei ce este apelată). În funcţie de tipul lor, obiectele pot avea diferite metode. Este posibil să existe metode de tipuri diferite fără a se crea confuzii. Este de asemenea posibil să creaţi obiecte şi metode folosind clasele, dar despre clase vom discuta mai târziu. Metoda `append` este definită pentru obiecte de tip listă, şi adaugă un element la sfârşitul listei. În acest exemplu ar fi echivalentă cu '`result = result + [b]`' dar mult mai eficientă.

4.7 Mai multe despre funcţii

Este posibil în Python să definim funcţii cu un număr variabil de argumente. Există trei modalitaţi de a realiza acest lucru, care pot fi combinate.

4.7.1 Valori implicite pentru argumente

Cea mai utilă metodă este să specificaţi valori implicite pentru argumente. În acest fel creaţi o funcţie care poate fi apelată cu mai puţine argumente decât a fost definită:

```
def ask_ok(prompt, retries=4, complaint='Da sau nu va rog!'):
        while 1:
                ok = raw_input(prompt)
                if ok in {'d', 'da'): return 1
                if ok in {'n', 'nu'): return 0
                retries = retries - 1
                if retries < 0: raise IOError, 'refusenik user'
                print complaint
```

Funcţia definită mai sus poate fi apelată în două moduri:

`ask_ok('Sunteţi sigur că doriţi să ieşiţi?')` sau: `ask_ok('Doriţi să ştergeţi fişierul?', 2)`

Valorile implicite sunt evaluate în timpul interpretării definiţiei funcţiei, deci următoarea secvenţă

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

va afişa `5`.

ATENŢIE!!!: Valorile implicite sunt evaluate o singură dată. Aveţi grijă atunci când valoarea implicită este un obiect modificabil. De exemplu funcţia următoare acumulează într-o listă argumentele ce îi sunt transmise în timpul unor apeluri consecutive:

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

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

va afişa

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

Dacă nu doriţi ca valoarea implicită să fie reţinută după terminarea funcţiei, puteţi proceda ca în exemplul următor:

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

4.7.2 Argumente de tip cuvinte cheie

Funcţiile pot fi apelate folosind în loc de argumente cuvinte cheie de forma: `cuvant_cheie = valoare`. Pentru exemplificare observaţi funcţia:

```
def parrot(voltage, state='mort', action='zboara', type='Norvegian Blue'):
        print "-- Acest papagal nu", action
        print "daca pui", voltage, "volti prin el."
        print "-- Doamne!", type
        print "-- E ", state, "!"
```

care poate fi apelată în mai multe moduri:

```
parrot(1000)
parrot(action='BOOM', voltage=10000)
parrot('o mie', state='apasa iarba')
parrot('un milion', 'fara viata', 'sare')
```

Dar următoarele apeluri ar genera erori:

```
parrot() # Lipseste argumentul obligatoriu
parrot(voltage=5.0, 'mort') # Dupa cuvantul cheie trebuie sa urmeze
un argument tip cheie
parrot(110, voltage=220) # Doua valori atribuite aceleia\c{s}i varibile
parrot(actor='John Cleese') # Cuvant cheie necunoscut
```

În general o listă de argumente trebuie să aibe oricâte argumente poziţionale, urmate de oricâte argumente de tip cuvinte cheie, unde cuvintele cheie trebuie alese din lista parametrilor formali. Nu este important dacă un parametru formal are o valoare implicită sau nu. Nici un argument nu poate primi o valoare de mai multe ori – numele de parametrii formali corespunzătoare argumentelor poziţionale nu pot fi folosite drept cuvinte cheie în cadrul aceluiaşi apel. Iată un exemplu cu erori datorate acestei reguli:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Atunci când unul dintre parametrii este de forma ***nume*, funcţia va primi o listă care va conţine parametrii de tip cuvânt cheie. Dacă se foloseşte un parametru de tipul **nume*, funcţia va primi o listă conţinând argumentele suplimentare în afara celor formale. Dacă sunt folosiţi împreună parametrul **nume* trebuie să se afle înaintea celui de tip ***nume*. Totul poate părea ambiguu dar veţi fi lămuriţi imediat:

```
def cheeseshop(kind, *arguments, **keywords):
        print " - Aveti", kind, "?"
        print " - Nu, nu avem", kind
        for arg in arguments: print arg
        print '-'*40
        for kw in keywords.keys(): print kw, ':', keywords[kw]
```

Un apel al acestei funcţii ar arăta astfel:

```
cheeseshop('Limburger',"Se vinde foarte repede, d-le"
        "Se vinde foarte, FOARTE repede, d-le"
        client='John Cleese'
        manager='Michael Palin'
        sketch='Cheese Shop Sketch')
```

Iar rezultatul va fi următorul:

```
- Aveti Limburger?
- Nu, nu avem Limburger
Se vinde foarte repede, d-le.
Se vinde foarte, FOARTE repede, d-le.
----------------------------------------
client : John Cleese
manager : Michael Palin
sketch : Cheese Shop Sketch
```

4.7.3 Liste de argumente arbitrare

O funcţie poate fi apelată cu un număr arbitrar de argumente. Înainte de argumentele arbitrare (opţionale) pot exista mai multe argumente normale:

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

4.7.4 Forme Lambda

La cererea utilizatorilor de Python, au fost adăugate câteva facilităţi specifice limbajelor funcţionale şi Lisp-ului.

Folosind cuvântul cheie `lambda` puteţi crea mici funcţii. Iată o funcţie care întoarce ca rezultat suma argumentelor: '`lambda a, b : a+b`'. Formele lambda pot fi folosite acolo unde sunt necesare funcţii obiect. Aceste funcţii sunt restrânse sintactic la o singură expresie:

```
>>> def make_incrementor(n):
...     return lambda x, incr=n: x+incr
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
>>>
```

4.7.5 Şirurile de documentaţie

Există anumite convenţii privind conţinutul şi formatarea şirurilor de documentaţie.

Prima linie ar trebui să fie scurtă şi foarte concisă. Pentru concizie nu ar trebui precizat numele obiectivului sau tipul, acestea fiind subînţelese. Această linie trebuie să înceapă cu literă mare şi să se termine cu virgulă.

Dacă există mai multe linii, cea de-a doua ar trebui să fie vidă, pentru a separa descrierea scurtă de cea amănunţită. Următoarele linii ar trebui să descrie mai pe larg obiectul, semnificaţia parametrilor, etc.

Interpretorul Python ia în consideraţie spaţiile din şirurile de caractere împărţite pe mai multe linii, din această cauză pentru a avea o documentaţie aranjată ”frumos” trebuie să vă folosiţi de un mic ”truc” la fel ca în exemplul următor:

```
>>> def my_function():
...     " " "Nu face decat documentarea.
...
...     Chiar nu face nimic!
...     " " "
...     pass
...
>>> print my_function.__doc__
Nu face decat documentarea.

        Chiar nu face nimic!
```

CAPITOL

CINCI

Strucuri de date

Acest capitol subliniază anumite lucruri deja cunoscute şi descrie altele noi.

5.1 Mai multe despre liste

Tipul de date listă mai dispune de câteva metode. Iată toate metodele obiectelor de tip listă:

`append(x)` Adaugă un element la sfârşitul listei.

`extend(L)` Adaugă la sfârşitul listei, elementele listei furnizate ca parametru.

`insert(i, x)` Inserează un element într-o anumită poziţie. Primul argument reprezintă indicele elementului din listă înaintea căruia se va face inserţia, deci `a.insert(0,x)` va insera elementul 'x' la începutul listei, iar `a.insert(len(a),x)` este echivalent cu `a.append(x)`.

`remove(x)` Şterge din listă primul element găsit cu valoarea 'x'. Dacă nu există un astfel de element apare o eroare.

`pop([i])` Şterge din listă elementul de pe poziţia 'i', şi întoarce valoarea acestuia. Dacă nu este specificat nici un parametru `a.pop()`, va şterge şi va returna ultimul element din listă.

`index(x)` Întoarce indicele primului parametru din listă care are valoarea 'x'.

`count(x)` Întoarce numărul de apariţii ale valorii 'x' între elementele listei.

`sort()` Sortează elementele listei.

`reverse()` Schimbă ordinea elementelor din listă.

Iată un exemplu care foloseşte majoritatea metodelor:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Folosirea listelor drept stive

Obiectele de tip listă pot fi folosite foarte uşor pentru a simula comportamentul unei stive. Pentru a adăuga un element pe stivă (PUSH), puteţi folosi `append()`. Pentru a scoate un element din stivă (POP), folosiţi `pop()` fără a specifica un index. Iată un exemplu:

```
>>> 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 Folosirea listelor drept cozi

Puteţi folosi listele foarte convenabil, pentru a implementa cozi. Spre deosebire de stive unde primul element adăugat este ultimul scos, la cozi primul element adăugat este primul scos. Pentru a adăuga un element folosiţi `append()`, iar pentru a extrage un element folosiţi `pop(0)`:

```
>>> queue = ["Eric", "Ion", "Mihai"]
>>> queue.append("Razvan")
>>> queue.append("Cristi")
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'Ion'
>>> queue
['Mihai', 'Razvan', 'Cristi']
```

5.1.3 Instrumente de programare funcţională

Există trei funcţii predefinite care sunt foarte utile în lucrul cu liste: `filter(), map(), reduce()`.

Funcţia '`filter()`' cu sintaxa '`filter(funcţie, secvenţă)`', întoarce o secvenţă formată din elementele secvenţei specificate ca parametru, care îndeplineşte condiţia testată de funcţie. Exemplul care urmează calculează numerele prime din intervalul 2, 25:

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

Funcţia `map()` cu sintaxa `map(funcţie, secvenţă)`, apelează funcţia specificată ca parametru pentru fiecare element din secvenţă, şi întoarce o nouă listă formată din rezultatele întoarse de funcţie. Pentru a calcula pătratele unor numere dintr-o listă puteţi folosi o secvenţă similară celei ce urmează:

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

Funcţia `map()` acceptă ca parametrii mai multe secvenţe. În acest caz funcţia transmisă ca parametru trebuie să fie modificată corespunzător, trebuie să accepte acelaşi număr de parametrii câte secvenţe sunt transmise. Dacă secvenţele sunt diferite ca lungime, atunci când una dintre secvenţe s-a terminat, în loc de un element funcţiei i se transmite `None`. Dacă în loc de un nume de funcţie, se transmite `None` funcţiei `map()`, atunci funcţia va fi înlocuită cu o funcţie care va întoarce ca rezultat parametrii primiţi. Puteţi folosi comportamentul funcţiei `map()` pentru a genera perchi de numere provenind din două liste:

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

Funcţia '`reduce(funcţie, secvenţă)`' întoarce o simplă valoare care este calculată în felul următor: este apelată funcţia (care este obligatoriu o funcţie binară ce acceptă numai 2 parametrii), cu parametrii primul şi al doilea termen al secvenţei, funcţia întoarce un rezultat care împreună cu al treilea element sunt transmise din nou funcţiei, care generează un alt rezultat şi aşa mai departe, până ce lista este epuizată.

Exemplul de mai jos calculează suma primelor numere naturale:

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

Dacă există un singur element în secvenţă valoarea acestuia va fi returnată. Dacă lista este goală, va fi generată o excepţie. Funcţia `reduce()` poate primi şi un al treilea parametru care semnifică valoarea de la care începe calculul. În acest caz primul apel al funcţiei are ca parametrii această valoare şi primul element al listei. Dacă funcţia `reduce()` este apelată cu trei parametrii, iar lista este goală, aceasta va întoarce ca rezultat al treilea parametru. Iată un exemplu care ilustrează modul de lucru al funcţiei `reduce()`:

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

5.1.4 Un alt mod de a genera liste

Există un mod de a crea liste mult mai concis decât prin intermediul funcţiilor `map(), filter()` sau `lambda()`. Definiţia listei este de cele mai multe ori mult mai clară decât cea obţinută prin alte metode. Acest mod generalizat, de a genera, a defini liste constă în asocierea unei expresii, cu o clauză `for` şi cu niciuna sau mai multe clauze `for` sau `if`. Rezultatul va fi o listă care provine din evaluarea expresiei în contextul clauzelor `for` şi `if` ce urmează. Dacă rezultatul evaluării expresiei va fi o pereche, atunci expresia trebuie parantezată corespunzător:

```
>>> 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]      # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [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]
```

5.2 Instrucţiunea `del`

Există o metodă de a şterge un element dintr-o listă specificând indicele elementului în loc de valoarea elementului. Această metodă poate fi folosită şi pentru a şterge porţiuni dintr-o listă. Toate acestea pot fi realizate utilizând instrucţiunea `del`:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` poate fi folosită şi pentru a şterge variabile, de exemplu o întreagă listă:

```
>>> del a
```

Vom vedea mai târziu şi alte modalităţi de a utiliza instrucţiunea `del`.

5.3 Perechi şi secvenţe

[1] Am obsevat până acum că listele şi şirurile de caractere au multe caracteristici comune, de exemplu: indexarea şi poziţionarea (slicing). Şirurile şi listele sunt două exemple de tipuri de date secvenţă. Deoarece Python este un limbaj care evoluează, în timp pot fi adăugate şi alte tipuri de date secvenţă. Există şi un alt tip standard de date: perechea (enumerarea).

O pereche, o enumerare este formată din mai multe valori separate prin virgule:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # enumerarile pot fi imbricate, combinate
... u = t, (1, 2, 3, 4 ,5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4 ,5))
```

După cum puteţi vedea perechile(enumerările) sunt afişate între paranteze astfel încât enumerările sau perechile îmbricate să fie interpretate corect. Perechile pot fi introduse cu sau fără paranteze, însă, adesea parantezele sunt necesare.

Perechile au multe utilizări. De exemplu : perechi de coordonate (x,y), înregistrările angajaţilor dintr-o bază de date, etc. Puteţi face o analogie (atâta timp cât rămâne numai o analogie menită să vă ajute la înţelegerea acestui tip de date) între tipurile de date compuse ce pot fi definite cu `struct` din C++, şi perechile din Python. Perechile, la fel ca şirurile nu pot fi modificate: nu pot fi atribuite valori unui anumit element al unei perechi (puteţi însă să simulaţi o modificare folosind alte metode). Este posibil să creaţi perechi care conţin obiecte modificabile, cum sunt listele de exemplu.

O problemă deosebită o constituie crearea perechilor cu zero sau un element. Există însă câteva "trucuri" de sintaxă care pot fi folosite pentru a rezolva această problemă. Perechile vide pot fi construite folosind o pereche de paranteze fără nimic între ele. O pereche cu un singur element poate fi construită specificând valoarea care va fi conţinută în pereche urmată de o virgulă. Cam neplăcut, dar funcţionează:

```
>>> empty = ()
>>> singleton = 'hello',        # <---- observati virgula
>>> len(empty)
0
>>> len(singleton)
1
('hello',)
```

Instrucţiunea `t = 12345, 54321, 'Salut!'` este un exemplu de împachetare. Valorile 12345, 54321 şi 'Salut!' sunt "împachetate" împreună. Operaţia inversă este de asemenea posibilă:

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

Această operaţie este numită suficient de elocvent "despachtarea unei secvenţe". Despachetarea unei secvenţe necesită ca lista de variabile de la stânga atribuirii să conţină tot atâtea variabile cât secvenţa împachetată de la dreapta. De observat este faptul că îmbricarea nu este decât o combinaţie succesivă de împachetări sau despachetări! Ingenios, nu?!

Există o asimetrie aici: în timp ce o împachetare va genera întotdeauna o pereche, despachtarea funcţionează pentru

[1] N.T. termenul românesc **PERECHE** este impropriu, forma originală fiind **TUPLE**.

orice fel de secvenţă.

5.4 Dicţionare

Un alt tip de date predefinit în Python şi care se poate dovedi foarte folositor este *dicţionarul*. Dicţionarele pot fi întâlnite şi în alte limbaje, sub alte nume, cum ar fi ”memorii asociative” sau ”valori asociative”. Spre deosebire de secvenţe (liste, şiruri, etc.) care sunt indexate cu numere (indicii sunt numere), dicţionarele sunt indexate de *chei*, care pot fi definite de oricare din tipurile de date invariabile(nemodificabile), de exemplu: şiruri de caractere sau numere. Perechile pot fi folosite drept chei ale unui dicţionar numai dacă conţin obiecte invariabile. Nu puteţi folosi drept chei listele deoarece acestea pot fi modificate folosind metode ca `append()` sau `extend()`.

Este indicat să vă gândiţi la un dicţionar ca la o mulţime neordonată de perechi cheie-valoare, cu observaţia că o cheie trebuie să fie unică într-un dicţionar. O pereche de acolade crează un dicţionar gol: {}. Puteţi crea un dicţionar dacă între acolade introduceţi o listă (ale cărei elemente sunt separate prin virgulă), de perechi cheie-valoare: `dicţionar = ’jack’:4098, ’Sape’:4139`. Operaţiile principale pe care le realizează un dicţionar sunt: stocarea unei valori cu anumită cheie şi extragerea unei valori cunoscându-se o anumită cheie. O pereche cheie-valoare poate fi ştearsă folosind instrucţiunea `del`. Dacă se adaugă o valoare în dicţionare pentru o cheie care există deja, valoarea veche asociată acelei chei este pierdută. O eroare este generată, bineînţeles, dacă veţi încerca să extrageţi o valoare pentru o cheie inexistentă.

Metoda `keys()` a unui obiect dicţionar întoarce o listă cu toate cheile existente în respectivul dicţionar. Lista returnată nu este sortată, însă pentru a o sorta puteţi folosi metoda `sort()` a obiectului listă returnat de funcţie. Pentru a verifica dacă o anumită cheie se află deja în dicţionar folosiţi metoda `has-key()`. Iată un mic exemplu:

```
>>> tel = {’jack’: 4098, ’sape’: 4139}
>>> tel[’guido’] = 4127
>>> tel
{’sape’: 4139, ’guido’: 4127, ’jack’: 4098}
>>> tel[’jack’]
4098
>>> del tel[’sape’]
>>> tel[’irv’] = 4127
>>> tel
{’guido’: 4127, ’irv’: 4127, ’jack’: 4098}
>>> tel.keys()
[’guido’, ’irv’, ’jack’]
>>> tel.has_key(’guido’)
1
```

5.5 Mai multe despre condiţii

Condiţiile folosite în cadrul instrucţiunilor `while` sau `if` pot conţine şi alţi operatori decât cei de comparaţie.

Operatorii de comparaţie `in` şi `not in` verifică dacă o anumită valoare se află sau nu într-o secvenţă dată. Operatorii `is` şi `is not` verifică dacă două obiecte sunt de fapt unul şi acelaşi obiect, acest lucru contează în cazul obiectelor modificabile cum sunt listele de exemplu.

Operatorii de comparaţie pot fi combinaţi. De exemplu `a < b == c` testează întâi dacă `a < b` iar apoi dacă `b` este egal cu `c`.

Condiţiile pot avea structuri foarte complexe dacă sunt folosiţi operatorii `and`, `or` sau `not`. Operatorii de comparaţie au aceeaşi prioritate care este mai mică decât aceea a operatorilor numerici. Operatorii logici, booleeni(`and,or,not`) au cea mai mică prioritate, mai mică decât aceea a operatorilor de comparaţie. Cea mai mare prioritate între operatorii logici o au `not`, apoi `and`, iar cea mai mică `or`. În concluzie “`A and not B or`

C” este echivalent cu “`A and (not B) or C`”. Bineînţeles puteţi folosi paranteze pentru a combina condiţiile şi pentru claritate.

Operatorii logici mai sunt denumiţi şi operatori “pe scurtătură” (shortcut operator), argumentele acestea sunt evaluate de la stânga la dreapta, şi evaluarea se opreşte odată ce rezultatul este determinat. De exemplu dacă A şi C sunt adevărate, iar B fals, expresia `A and B and C` nu este evaluată până la sfârşit, deci expresia C nu este evaluată pentru că ar fi inutil.

Este posibilă atribuirea unei valori rezultate dintr-o comparaţie sau orice altă condiţie, unei variabile:

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

Spre deosebire de C, în Python nu sunt permise atribuirii în cadrul expresiilor. Programatorii de C pot critica acest lucru, dar prin aceasta se evită o mulţime de erori întâlnite în programele C, de exemplu = în loc de ==.

5.6 Compararea secvenţelor

Obiectelor de tip secvenţă pot fi comparate cu obiecte secvenţă de acelaşi tip. Comparaţiile între secvenţe folosesc principiile ordonării *lexicografice*: sunt comparate întâi primele două elemente din fiecare listă, dacă diferă rezultatul este afişat, dacă sunt egale se trece la compararea următoarelor două elemente, şi aşa mai departe până la epuizarea uneia dintre liste. Dacă elementele ce urmează a fi comparate sunt la râdul lor liste, compararea lexicografică are loc recursiv. Dacă toate elementele celor două liste sunt egale se consideră că listele sunt egale. Dacă una dintre secvenţe este o subsecvenţăa iniţială a celeilalte, atunci secvenţă mai scurtă este cea mai mică. Ordonarea lexicografică pentru şiruri foloseşte ordinea ASCII a caracterelor. Exemple de comparaţii între secvencţe de acelaşi tip:

```
(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)
```

Se pot compara obiecte de tipuri diferite. Rezultatul este însă arbitrar: tipurile fiind ordonate după numele lor. În concluzie o listă va fi întotdeauna mai mică decât un şir, un şir mai mic decât o pereche, etc. Comparaţiile între numere se fac luând în consideraţie valoarea numerică chiar dacă numerele sunt de tipuri diferite, şi ca atare 0 va fi egal cu 0.0, etc. [2]

[2] Regulile de comparaţie între obiecte de tipuri diferite nu ar trebui considerate de foarte mare încredere, deoarece acestea pot fi schimbate la versiuni viitoare ale limbajului.

CAPITOL

ŞASE

Module

Dacă ieşiţi din interpretorul Python şi intraţi din nou, definiţiile pe care le-aţi făcut (funcţii, varibile, etc.) se pierd. Dacă aveţi de gând să scrieţi un program mai lung, ar fi mai bine să folosiţi un editor de texte pentru a scrie programul într-un fişier, şi apoi să rulaţi interpretorul Python cu parametru fişierul în care se află programul. Fişierul în care se află programul se numeşte *script*. Pe măsură ce programul se va mări veţi simţi nevoia să îl împărţiţi în mai multe fişiere. S-ar putea la fel de bine să doriţi să folosiţi o funcţie în mai multe programe fără a fi nevoit să copiaţi definiţia funcţiei în fiecare program.

În Python puteţi scrie anumite definiţii într-un fişier, pe care îl puteţi include în alte programe. Un astfel de fişier se numeşte **modul**. Definiţiile dintr-un modul pot fi importate în alte module, sau în modulul principal (programul iniţial). Un modul este un fişier care conţine definiţii şi instrucţiuni Python. Numele fişierului în care se află un anumit modul este dat de numele modulului şi extensia ”.py” la sfârşit. Într-un modul, numele acestuia este accesibil prin intermediul variabilei globale `__name__`. Folosiţi acum editorul dumneavoastră de texte preferat pentru a creea fişierul ”fibo.py”:

```
# modul cu numerele lui Fibonacci

def fib(n):     # scrie \c{s}irul lui Fibonacci pana la n
        a, b = 0, 1
        while b < n:
                print b,
                a, b = b, a+b

def fib2(n):    # intoarce \c{s}irul lui Fibonacci pana la n
        result = []
        a, b = 0, 1
        while b < n:
                result.append(b)
                a, b = b, a+b
        return result
```

Acum deschideţi interpretorul Python şi tastaţi comanda:

```
>>> import fibo
```

Această instrucţiune nu încarcă în tabela de simboluri numele funcţiilor definite în modulul fibo, ci numai numele modulului. Folosind numele modulului puteţi accesa funcţiile definite în interiorul acestuia:

```
>>> 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'
```

În cazul în care vă deranjează aceste nume lungi puteţi proceda în felul următor:

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

6.1 Mai multe despre module

Un modul poate conţine atât instrucţiuni executabile cât şi definiţii de funcţii. Aceste instrucţiuni sunt menite să realizeze iniţializarea modulului şi se execută o singură dată atunci când modulul este importat.

Fiecare modul are propria sa tabelă de simboluri care este folosită de funcţiile definite în modul ca o tabelă de simboluri globală.

Cu toate astea, autorul unui modul se poate folosi de variabilele globale, fără a-şi face griji pentru eventuale coliziuni cu variabilele globale ale mediului în care modulul va fi importat. Pe de altă parte dacă aveţi neapărat nevoie puteţi modifica direct variabilele globale ale unui modul, folosind aceeaşi convenţie ca la apelarea funcţiilor unui modul: `modul.variabilă`.

Modulele pot importa alte module. Instrucţiunile `import` pot fi plasate la începutul programului şi este indicat, dar nu este obligatoriu.

Există o metodă de a încărca definiţiile unui anumit modul direct în tabela de simboluri globală a modulului care importă:

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

În exemplul de mai sus, numele modulului nu a fost introdus în tabela de simboluri locală. Pentru a încărca toate definiţiile modulului direct în tabela locală folosiţi:

În exemplul de mai sus, în tabela locală au fost încărcate toate numele de variabile şi funcţii, mai putin cele care încep cu caracterul "_".

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

6.1.1 Calea în care sunt căutate modulele

Atunci când un modul numit `xxx` este importat, interpretorul caută un fişier numit 'xxx.py' în directorul curent, şi în toate directoarele specificate în variabila de sistem PYTHONPATH. Aceasta are aceeaşi sintaxa variabila de sistem

PATH, care este o listă a directorului. Dacă variabila PYTHONPATH nu este definită modulul va fi căutat în directorul implicit de instalare, de obicei '/usr/local/lib/python'.

6.1.2 Fişiere Python "compilate"

Un important aport de viteză la pornirea programelor care încarcă o mulţime de module standard este adus de următorul comportament al Python-ului: dacă în directorul unde este găsit fişierul 'xxx.py' se mai află şi 'xxx.pyc', se presupune că acest ultim fişier conţine o variantă compilată a modulului şi este încărcat în loc de 'xxx.py'. Timpul la care a fost modificat ultima oară fişierul '.py' este înregistrat în fişierul '.pyc' în momentul în care este creat, iar în momentul importului dacă aceste momente de timp nu corespund fişierului '.pyc' este ignorat.

De obicei nu trebuie să faceţi nimic deosebit pentru a crea fişierul 'xxx.pyc'. Oridecâteori 'xxx.py' este compilat cu succes este creat sau suprascris şi fişierul 'xxx.pyc'. Dacă scrierea fişierului compilat nu reuşeşte veţi fi avertizat printr-un mesaj de eroare. Conţinutul fişierelor de tip '.pyc' este independent de platformă, astfel încât Python devine un limbaj foarte portabil.

Câteva indicaţii pentru experţi:

- Când interpretorul Python este apelat folosind opţiunea **-O**, se generează cod optimizat care va fi stocat în fişiere de tip '.pyo'. La momentul actual optimizatorul nu este de prea mare folos, nu face decât să elimine din cod instrucţiunile `assert` şi `SET-LINENO`. Atunci când se foloseşte această opţiune tot codul este optimizat, fişierele '.pyc' sunt ignorate iar fişierele '.py' sunt compilate şi optimizate.
- Dacă interpretorul Python este lansat în execuţie folosind opţiunea **-O**, compilatorul poate genera în anumite cazuri cod compilat eronat. Versiunea curentă elimină şirurile `__doc__` din codul compilat, astfel încât acesta să fie mai compact. Din moment ce unele programe pot folosi aceste şiruri, eliminarea lor ar putea genera erori. În concluzie, ar trebui să folosiţi această opţiune numai atunci când ştiţi foarte bine ce faceţi.
- Un program nu funcţionează mai rapid, atunci când este citit dintr-un fişier de tip '.pyc' sau `.pyo`, faţă de unul '.py'. Singurul lucru care diferă este viteza de încărcare a programelor în memorie.
- Atunci când un script este rulat, prin lansarea interpretorului şi parametru numele fişierului în care se află script-ul, codul script-ului nu este compilat şi nu se generează fişiere de tip '.pyc' sau '.pyo'. În cazul în care aveţi un script de dimensiuni mari, şi doriţi să creşteţi viteza de încărcare, puteţi să mutaţi bucăţi din cod într-un modul pe care să îl importaţi apoi în script. Interpretorul Python poate fi lansat şi folosind direct un fişier cu extensia '.pyc' sau '.pyo'.
- Este posibil să lansaţi interpretorul Python doar cu un fişier de tip '.pyc' (sau '.pyo' dacă se foloseşte **-O**), fără ca fişierul '.py' asociat să existe. Puteţi folosi această facilitate atunci când distribuiţi o bibliotecă şi nu doriţi să fie modificată.
- Modulul `compileall` poate fi folosit pentru a crea fişiere '.pyc' sau '.pyo' pentru toate modulele dintr-un director.

6.2 Module standard

Python dispune de o bilbliotecă de module standard, a căror descriere o puteţi găsi îndocumentul "The Python Library Reference". Unele module sunt integrate în interpretor, deşi nu fac parte din nucleul limbajului, din definiţia acestuia, dar sunt integrate pentru eficienţă sau pentru a facilita accesul la primitivele sistemului de operare. Setul de module integrate este o optiune de configurare, care de asemenea depinde de particularitatea platformei. De exemplu modulul `amoeba` este disponibil pe sisteme care pun la dispoziţie primitive Amoeba. Un anumit modul necesită o atenţie specială: `sys` care este integrat în orice interpretor Python. Variabilele `sys.ps1` şi `sys.ps2` definesc şirurile de caractere ce vor fi folosite de interpretor pentru prompt-ul principal şi cel secundar:

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

Aceste două variabile sunt definite doar dacă interpretorul se află în mod interactiv. Variabila `sys.path` conţine o lisă de directoare unde Python caută module. Această variabilă este iniţializată din variabila de sistem PYTHONPATH, sau cu o valoare implicită în cazul în care PYTHONPATH nu este definită. Puteţi modifica `sys.path` folosind operaţii specifice listelor:

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

6.3 Funcţia `dir()`

Funcţia integrată `dir()` poate fi folosită pentru a determina ce nume(de variabile, funcţii, etc.) defineşte un modul. Rezultatul acestei funcţii este o listă sortată de şiruri de caractere:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

Dacă funcţia `dir()` este apelată fără argumente, va lista numele ce sunt definite (local) până în momentul curent:

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

Observaţi că listează diferite tipuri de nume: variabile, funcţii, module, etc.

`dir()` nu listează şi numele de funcţii sau variabile integrate. Dacă doriţi o astfel de listă, puteţi apela `dir(__builtin__)`, unde `__buildin__` este modulul în care se află funcţiile şi variabilele integrate:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

6.4 Pachete

Pachetele sunt o modalitate prin care Python structurează modul de acces la module(şi la definiţiile existente în module). Pentru exemplificare numele modulului `A.B` desemnează un submodul '`B`' definit în cadrul pachetului '`A`'.

Să presupunem că doriţi să proiectaţi o colecţie de module (un pachet) pentru manipularea fişierelor de sunet, şi a sunetelor. Există o multitudine de formate de sunet (de obicei recunoscute după extensia lor '.wav', '.aiff', '.au') aşa că este necesară creearea şi întreţinerea unei colecţii de module încontinuă creştere care să permită conversia între diverse formate de fişiere. Există multe alte operaţii pe care poate aţi dori să le executaţi (mixaje, adăugarea ecoului, efecte stereo, etc.), aşa că, de fapt, veţi scrie o serie nesfârşită de module care să permită aceste operaţii. Aveţi mai jos o posibilă structură a pachetului dumneavoastră (în sensul unei ierahii de fişiere):

```
Sound/                          Pachet "tata"
      __init__.py               Initializarea pachetului 'sound'
      Formats/                  Subpachet pentru conversii intre fi\c{s}iere
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Subpachet pentru efecte acustice
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Subpachet pentru filtre
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Fişierele de tipul '__init__.py' sunt necesare pentru a face Python să trateze directoarele ca pachete. Acestea sunt necesare pentru a preveni situaţia ca un director cu un nume comun, de exemplu `string`, să ascundă un modul valid cu acelaşi nume. În cel mai simplu caz '__init.py__' poate fi un fişier gol, dar poate conţine şi cod de iniţializare.

Utilizatorii unui pachet pot importa doar un anumit modul din cadrul pachetului:

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

Instrucţiunea de mai sus a încărcat modulul `Sound.Effects.echo`, dar o funcţie din cadrul modulului trebuie aplelată folosindu-i numele întreg.

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

O alternativă pentru a importa un modul este:

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

În acest moment numele definite în modul au fost încărcate în tabela locală de simboluri, deci numele de funcţii şi variabile sunt disponibile fără prefix:

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

O altă variantă este încărcarea (importarea) directă a funcţiei sau variabilei dorite:

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

Un exemplu de utilizare a funcţiei `echofilter()`:

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

De observat că atunci când se foloseşte sintaxa `from` *packet* `import` *element*, elementul poate fi, fie un submodul (sau subpachet), fie un nume definit în pachet, de exemplu un nume de funcţie sau variabilă, sau un nume de clasă.

Instrucţiunea `import` testează întâi dacă elementul este definit sau nu în pachet. Dacă elementul nu este definit, Python presupune că este vorba de un modul şi încearcă să-l încarce. Dacă încărcarea nu reuşeşte este generată o excepţie de tipul `ImportError`.

Când folosiţi o sintaxă de genul: `import element.subelement.subsubelement`, fiecare element, mai puţin ultimul trebuie obligatoriu să fie un pachet. Ultimul element poate fi un modul, un pachet, dar nu poate fi o clasă, o funcţie, o variabilă, etc. definită în elementul anterior.

6.4.1 Importarea tuturor modulelor dintr-un pachet

Ce se va întâmpla dacă un programator va folosi o instrucţiune de genul: `from Sound.Effects import *`? Într-un caz ideal, programatorul ar putea să spere că interpretorul Python va realiza o căutare recursivă în sistemul de fişiere şi directoare după modulele existente în pachet. Din nefericire nu funcţionează foarte bine pe platforme Mac sau Windows, din varii motive. Pe astfel de platforme existenţa unui fişier 'ECHO.PY' nu garantează că în acest fişier se află un modul numit `Echo`, cât şi `echo` sau `ecHo`, şi aşa mai departe. Spre exemplu Windows 95 are prostul obicei de a afişa numele de fişiere cu prima literă mare. Sistemul de fişiere DOS 8+3 ridică o altă piedică în calea numelor lungi de module.

Singura soluţie pentru astfel de probleme este ca autorul pachetului să specifice exact ce module există în pachet. Instrucţiunea `import()` foloseşte următoarea convenţie: dacă codul existent în fişierul '__init__.py' defineşte o listă cu numele `__all__`, aceasta este considerată lista cu modulele ce ar trebui încărcate la execuţia unei instrucţiuni `from` *pachet* `import *`. Este obligaţia autorului pachetului să modifice lista atunci când este cazul. Autorii de pachete pot lua de asemenea decizia de a nu defini această listă, dacă nu vor ca programatorul care foloseşte pachetul să execute instrucţiuni de genul `import *`. Fişierul 'Sounds/Effects/__init__.py' ar trebui să conţină următoarele:

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

În acest caz instrucţiunea `from Sound.Effects import *` ar încărca cele trei submodule ale modulului `Sound`.

Dacă variabila `__all__` nu este definită, instrucţiunea de import nu va importa toate submodulele ci va importa numai pachetul `Sound.Effects` (eventual va executa codul de iniţializare din '__init__.py') şi apoi va importa toate numele definite în pachet. Se vor importa şi submodulele pachetului dacă acest lucru este specificat prin instrucţiuni `import`. Priviţi următorul cod :

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

În exemplul anterior, modulele echo şi surround sunt importate deoarece acest lucru este specificat (prin instrucţiuni `import`) în cadrul pachetului `Sound.Effects`.

Nu este foarte eficient să folosiţi instrucţiuni de tipul `import *`, pentru că în acest fel codul dumneavoastră va deveni neclar, nefiind clar care anume module sunt încărcate. Totuşi dacă aţi lansat interpretorul în mod interactiv, puteţi folosi astfel de instrucţiuni pentru a evita scrierea multor instrucţiuni.

În concluzie nu este nimic greşit în folosirea unei instrucţiuni similare cu `from` *Packet* `import` *submodul*, de fapt aceasta este notaţia recomandată.

6.4.2 Referenţieri între pachete

Apare deseori necesitatea ca într-un submodul să apară un alt submodul. De exemplu modulul `surround` s-ar putea folosi de modulele `echo`. De fapt astfel de referinţe sunt atât de întâlnite încât instrucţiunea `import` examinează întâi pachetul "cel mai cuprinzător", şi apoi caută în lista de directoare. Cu toate astea modulul `surround` poate folosi mult mai simplu o instrucţiune `import echo` sau `from echo import` *echofilter*. Dacă modulul ce se doreşte importat nu se află în pachetul curent, instrucţiunea `import` caută "mai sus" în ierarhia de modul a pachetului.

Atunci când pachetele sunt strucuturate în subpachete, nu există o modalitate de a prescurta referirile la alte submodule, ci trebuie folosite numele întregi ale subpachetelor. De exemplu dacă modulul `Sound.Filters.vocoder` are nevoie de modulul `echo` din pachetul `Sound.Effects`, poate folosi instrucţiunea: `from Sound.Effects import echo`.

CAPITOL

ŞAPTE

Intrări şi ieşiri

Există câteva modalităţi de a prezenta rezultatele unui program. Datele pot fi afişate într-un format care poate fi înţeles de om, sau pot fi scrise într-un fişier pentru a putea fi prelucrate mai târziu. Acest capitol va explica câteva dintre posibilităţi.

7.1 Formatarea mai elegantă a datelor la ieşire

Până acum am întâlnit două metode de a afişa valori: instrucţiunea `print` şi expresii. (O a treia metodă este folosind metoda `write()` a obiectelor de tip fişier. Fişierul standard de ieşire este referit ca ‘sys.stdout’.)

Adesea veţi dori să aveţi mai mult control asupra modului de afişare a valorilor. Există două metode pentru a controla modul de afişare: prima este să modificaţi singur un şir de caractere, folosind diversele operaţii existente, iar apoi să îl afişaţi. Modulul `string` conţine câteva operaţii utile pentru manipularea şirurilor de caractere. O a doua metodă este folosirea operatorului %, cu un şir, ca argument stânga. Operatorul % interpretează argumentul stânga în acelaşi mod ca şi şirul de formatare al funcţiei C `sprintf()` aplicându-l asupra argumentului din dreapta şi returnând şirul rezultat în urma ecestei formatări. O singură întrebare rămâne: cum pot fi convertite valorile în şiruri de caractere? Din fericire Python poate converti orice tip de valoare în şir de caractere: fie prin funcţia `repr()`, fie scriind valoarea între apostroafe (“). Iată câteva exemple:

Până acum am întâlnit două metode de a afişa valori: instrucţiunea `print` şi expresii. (O a treia metodă este folosind `write()` a obiectelor de tip fişier. Fişierul standard de ieşire este referit ca `sys.stdout`.)

```
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Valoarea lui x este ' + 'x' + ', \c{s}i a lui y este ' + 'y' + '...'
>>> print s
Valoarea lui x este 32.5, \c{s}i a lui y este 40000...
>>> # Apostroafele inverse operea\u{a} \^{i}n alt fel asupra numerelor :
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[32.5, 40000]'
>>> # Convertirea unui \c{s}ir ad\u{a}ug\^{a}nd apostroafe \c{s}i backslashe-uri:
... hello = 'hello, world\n'
>>> hellos = 'hello'
>>> print hellos
'hello, world\n'
>>> # Argumentele unor apostroafe inverse pot fi perechi ( tuple ) :
... 'x, y, ('spam', 'eggs')'
"(32.5, 40000, ('spam', 'eggs'))"
```

Iată două modalităţi de a genera un tabel cu pătratele şi cuburile numerelor naturale:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(‘x‘, 2), string.rjust(‘x*x‘, 3),
...     # Observati ultimul caracter "," de pe ultima linie
...     print string.rjust(‘x*x*x‘, 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
```

Observaţi că între coloane a fost adăugat un spaţiu. Acest comportament este asigurat de modul în care lucrează instrucţiunea `print`: întotdeauna adaugă un spaţiu între argumentele sale.

Acest exemplu demonstrează utilizarea funcţiei `string.rjust()`, care aliniază la dreapta un şir de caractere, într-un câmp de o anumită dimensiune dată, introducând spaţii la stânga şirului. Există şi alte funcţii similare `string.ljust()`, `string.center()`. Aceste funcţii nu afişează nimic, nu fac decât să returneze un alt şir de caractere. Dacă şirul primit este mai lung, aceste funcţii nu îl modifică, ci îl returnează intact. Acest mecanism probabil că vă va strica aranjarea pe coloane, dar este o variantă preferabilă celeilalte, adică trunchierea şirului. Dacă doriţi să truchiaţi un şir, puteţi oricând să folosiţi operaţiile de porţionare (slicing), ca de exemplu: `string.ljust(x,n)[0:n]`.

Mai există o funcţie utilă, care ”umple” cu zero-uri un şir, adăugându-le la stânga şirului original, până când acesta ajunge la o anumită dimensiune. Această funcţie este `string.zfill()`:

```
>>> import string
>>> string.zfill(’12’, 5)
’00012’
>>> string.zfill(’-3.14’, 7)
’-003.14’
>>> string.zfill(’3.14159265359’, 5)
’3.14159265359’
```

Folosind operatorul % ar arăta astfel:

```
>>> import math
>>> print 'Valoarea lui PI este aprox. %5.3f.' % math.pi
Valoarea lui PI este aprox. 3.142.
```

Dacă există mai multe formatări în şir, trebuie să se transmita ca operator dreapta o pereche (tupla), astfel :

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

Majoritatea tipurilor de formate funcţionează exact ca în C, şi nu necesită decât transmiterea corectă a operandului din dreapta. Nerespectarea acestei reguli va genera o excepţie. Specificatorul de format `%s` este mai flexibil, dacă parametrul asociat din partea dreaptă nu este de tip şir de caractere, va fi automat convertit la şir de caractere folosind funcţia integrată `str()`. Specificatorii de format `%u` şi `%p` din C nu sunt acceptaţi şi de Python.

Dacă aveţi un şir lung pe care nu doriţi să-l împărţiţi în mai multe şiruri, ar fi interesant să vă puteţi referi la variabile prin nume în loc de poziţie. Acest lucru poate fi realizat folosind modalitatea `numevariabilă format`, ca înurmatorul exemplu :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Acest lucru se poate dovedi foarte util atunci când doriţi să afişaţi variabilele predefinite folosind funcţia `vars()` care întoarce un dicţionar cu toate variabilele locale.

7.2 Fişiere

Funcţia `open()` are ca rezultat un obiect de tip fişier, şi este de cele mai multe ori apelată cu doi parametrii: numele de fişier şi modul de acces la fişier:

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

Primul argument este un şir de caractere care conţine numele fişierului ce urmează să fie deschis. Al doilea argument este tot un şir de caractere ce conţine doar câteva caractere ce descriu modul în care fişierul va fi utilizat. Modul poate fi:

- 'r' = fişierul va putea fi numai citit
- 'w' = fişierul va putea decât să fie scris (în cazul în care un fişier există deja, acesta va fi suprascris)
- 'a' = fişierul va fi deschis pentru actualizare (toate datele scrise vor fi adăugate la sfârşitul fişierului)
- 'r+' = în fişierul ce va fi deschis se pot executa atât operaţii de scriere cât şi de citire.

Pe Windows şi Macintosh adăugarea caracterului 'b' la sfârşitul şirului prin care se specifică modul de acces, indică interpretorului Python să deschidă fişierul în mod binar. Există deci modurile de acces 'rb, 'wb', 'r+b'. Windows face distincţie între fişierele de tip text şi cele binare: caracterele de sfârşit de linie sunt modificate atunci când se scriu sau se citesc date. Aceaste modificări "din spatele scenei" sunt binevenite în cazul fişierelor text, dar nu pot face decât rău în cazul fişierelor binare cum sunt `.JPG` sau `.EXE` de exemplu. Aveţi deci grijă să folosiţi modul binar când lucraţi cu astfel de fişiere.

7.2.1 Metodele obiectelor fişier

Exemplele acestei secţiuni vor presupune că un obiect 'f' de tip fişier a fost deja creat, deci fişierul a fost deja deschis.

Pentru a citi conţinutul unui fişier apelaţi `f.read(dimensiune)`, care citeşte o cantitate de date şi o returnează ca string. Parametrul dimensiune este opţional. Atunci când acest parametru este omis sau este negativ, întregul fişier va fi citit şi returnat ca şir de caractere. Apare o problemă bineînţeles dacă memoria maşinii dumneavoastră este mai mică decât dimensiunea fişierului. Dacă parametrul dimensiune este transmis funcţiei din fişier vor fi citiţi cel mult atâţia bytes câţi sunt specificaţi prin acest parametru.

Dacă s-a ajuns la sfârşitul fişierului `f.read()` va returna un şir vid(" "):

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

Metoda `f.readline()` citeşte o singură linie din fişier. Un caracter "linie nouă"(newline), "\n" este adăugat la sfârşitul fiecărui şir. Acest caracter este omis dacă este vorba despre ultima linie din fişier şi dacă acesta nu se termină cu un caracter "linie nouă". Toate acestea fac rezultatul neclar. Dacă rezultatul este un şir gol, atunci a fost atins sfârşitul fişierului, în timp ce dacă rezultatul este doar caracterul "\n" înseamnă că din fişier a fost citită o linie goală:

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Metoda `f.readlines()` a obiectelor de tip fişier, întoarce o listă conţinând toate liniile din fişier. Dacă metoda este apelată cu argumentul dimensiune, atunci din fişier sunt citiţi atâţia bytes câţi sunt specificaţi prin acest parametru şi încă atâţia bytes câţi sunt necesari pentru a completa o linie. Această metodă este folosită pentru citirea eficientă pe linii a fişierelor de dimensiuni mari fără a întâmpina dificultăţi cu memoria. Vor fi returnate numai linii complete:

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

Metoda `write(`*şir)* scrie conţinutul şirului de caractere în fişier, întorcând `None` ca rezultat:

```
>>> f.write('This is a test\n')
```

`f.tell()` are ca rezultat un număr întreg reprezentând poziţia cursorului în fişier, poziţie măsurată în bytes faţă de începutul fişierului. Pentru a schimba poziţia cursorului folosiţi funcţia `f.seek(deplasare,referinţă)`.

Noua poziţie este calculată în felul următor: cursorul va fi deplasat cu `deplasare` bytes, faţă de începutul fişierului dacă `referinţă` este 0, faţă de poziţia curentă a cursorului dacă `referinţă` este 1 şi faţă de sfârşitul fişierului dacă `referinţă` este 2. Dacă al doilea parametru este omis, valoarea lui implicită va fi 0, deci punctul de referinţă va fi începutul fişierului:

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 5th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

Când terminaţi lucrul cu un fişier, acesta trebuie închis folosind `f.close()`. După închiderea fişierului orice încercare de operaţie asupra fişierului va eşua:

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

Obiectele fişier poseadă şi alte metode, cum ar fi `isatty()` şi `truncate()` care sunt mai puţin folosite. (Consultaţi "Python Library Refference" pentru mai multe detalii).

7.2.2 Modulul `pickle`

Şirurile de caractere pot fi citite şi scrise foarte uşor dintr-un, respectiv într-un fişier. Cu numerele lucrurile se complică puţin. Aţi putea să transformaţi înainte de scrierea în fişier, numărul în string, apoi să îl scrieţi, iar la citire să îl transformaţi înapoi în număr, dar un astfel de mecanism este complet ineficient.

Pentru aceste situaţii, şi altele mult mai complicate Python dispune de modulul `pickle` care poate transforma orice obiect Python într-un string. Acest proces se cheamă **pickling**, denumirea procesului invers se numeşte **unpickling**. Între aceste două proceduri string-ul poate fi salvat într-un fişier, transmis în reţea, etc.

Cea mai simplă metodă de a salva un obiect într-un fişier este următoarea:

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

Iar pentru a încărca un obiect dintr-un fişier:

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

Există şi alte metode de a transforma un obiect într-un şir fără a-l salva într-un fişier. Pentru mai multe detalii consultaţi “Python Library Refference”.

Prin procedeele de ‘pickling’ şi ‘unpickling’, `pickle` poate stoca obiecte ce pot fi apoi reutilizate. Termenul tehnic pentru un astfel de obiect este *obiect persistent*. Deoarece aceste metode sunt foarte des folosite, programatorii care creează extensii ale limbajului Python au grijă ca tipurile de date nou definite să poată fi salvate şi încărcate corect(mai bine zis să se comporte corect în procesul de ‘pickling’ şi apoi ‘unpickling’).

CAPITOL

OPT

Erori şi excepţii

Până acum erorile au fost doar menţionate, dar dacă aţi încercat exemplele prezentate probabil aţi şi întâlnit câteva dintre ele. Există (cel puţin) două categorii de erori: erori de sintaxă şi excepţii.

8.1 Erori de sintaxă

Erorile de sintaxă sunt cel mai des întâlnite erori atâta timp cât sunteţi un începător în limbajul Python:

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

Interpretorul reproduce linia care a cauzat eroarea şi afişează o săgeată în dreptul instrucţinii care a generat eroarea. Eroarea este cauzată (sau cel puţin detectată) de instrucţiunea dinaintea săgeţii. În exemplul de mai sus eroarea este generată de instrucţiunea `print`, deoarece înaintea acestei instrucţiuni ar trebui să existe un caracter ':'. Numele fişierului şi numărul liniei care a generat eroarea sunt afişate, astfel încât dacă eroarea provine dintr-un script să puteţi corecta cât mai comod.

8.2 Excepţii

Chiar dacă o expresie sau o instrucţiune este corectă din punct de vedere sintactic, aceasta poate genera o eroare în momentul în care este executată. Erorile generate (detectate) în timpul execuţiei se numesc excepţii, şi nu sunt neapărat fatale. Veţi întreba cum puteţi evita astfel de erori utilizând limbajul Python.

Majoritatea excepţiilor nu sunt tratate de program şi generează mesaje de eroare ca în exemplul de mai jos:

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

Ultima linie a mesajului de eroare indică ce s-a întâmplat. Excepţiile sunt de diferite tipuri, iar tipul excepţiei este de asemenea afişat în corpul mesajului de eroare. În exemplul anterior: `ZeroDivisionError`, `NameError` şi `TypeError`, şirul afişat ce desemnează tipul excepţiei este un nume predefinit pentru respectivul tip de excepţie. Acest lucru este valabil pentru toate excepţiile predefinite, dar se pot defini şi alte tipuri de excepţii.

A doua parte a liniei reprezintă detaliile excepţiei, descriind mai bine ce s-a întâmplat.

În *Python Library Reference* sunt listate excepţiile implicite şi semnificaţiile lor.

8.3 Tratarea excepţiilor

Este posibil să scrieţi programe care tratează anumite tipuri de excepţii . În următorul exemplu este implementată o buclă care cere utilizatorului introducerea unui număr. Bucla este întreruptă în momentul în care utilizatorul introduce un număr corect, astfel procesul de introducere continuă. Procesul poate fi întrerupt folosind combinaţia de taste 'CTRL-C'. Dacă utilizatorul foloseşte această combinaţie de taste, programul este întrerupt, dar înainte este generată excepţia: `KeyboardInterrupt`:

```
>>> while 1:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number.  Try again..."
...
```

Instrucţiunea `try` funcţionează în felul următor:

- Întâi sunt executate instrucţiunile din blocul `try` (blocul de instrucţiuni dintre instrucţiunile `try` şi `except`).
- Dacă nu este generată nici o excepţie, instrucţiunile din blocul `except` nu sunt executate şi programul continuă.
- Dacă apare o excepţie în timpul execuţiei instrucţiunilor din blocul `try` şi dacă tipul excepţiei este acela pe care îl tratează şi blocul `except`, atunci instruţiunile din acest bloc nu sunt executate. După ce excepţia este tratată, execuţia continuă, nemai executându-se instrucţiunile rămase din blocul `try` (adică instrucţiunile ce urmează după instrucţiunea care a generat excepţia).
- Dacă excepţia nu este prevăzută între excepţiile tratate în blocul `except`, ea este transmisă altor structuri `try`.Dacă nu este găsit un bloc `except` care să trateze excepţia, programul este întrerupt, iar respectiva excepţie se numeşte `excepţie netratată` (`unhandled exception`).

O instrucţiune `try` poate avea mai multe clauze `except`, implementând astfel mai multe tipuri de 'tratament' pentru mai multe tipuri de excepţii. Dintre toate clauzele `except` va fi executată cel mult una. Instrucţiunile unei clauze `except`, tratează numai excepţiile generate în blocul `try` căruia clauza îi este asociată, nu şi excepţiile ce pot apare în alte clauze `except`. O astfel de clauză poate trata mai multe tipuri de excepţii desemnate printr-o lista inchisă între paranteze, ca de exemplu :

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

Ultima clauză `except` poate fi folosită fără a se specifica ce anume excepţie tratează, în acest caz această clauză tratează toate excepţiile netratate de celelalte clauze. Puteţi folosi această ultimă precauţie, şi este indicat să o faceţi, pentru a ascunde toate erorile ce pot apărea şi pe care nu le-aţi anticipat. O puteţi de asemenea folosi pentru a afişa un mesaj de eroare şi apoi a regenera excepţia, care va fi transmisă mai departe, urmând să fie eventual tratată de un alt bloc `try`:

```
import string, sys
    try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(string.strip(s))
    except IOError, (errno, strerror):
        print "I/O error(%s): %s" % (errno, strerror)
    except ValueError:
        print "Nu poate fi convertit \^{i}ninteger"
    except:
        print "Eroare neasteptata:", sys.exc_info()[0]
        raise
```

Instrucţiunile `try` dispun şi de o clauză opţională: `else`. Atunci când aceasta este folosită, ea trebuie să fie ultima dintre clauze. Instrucţiunile acestei clauze vor fi executate atunci când, blocul `try` nu generează nici o excepţie. Iată un exemplu:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Utilizarea clauzei `else` este mai adecvată decât adăugarea unor linii suplimentare de cod la sfârşitul blocului `try`, pentru că în acest fel se evită detectarea unei excepţii care nu a fost generată de instrucţiunile blocului `try`.

O excepţie poate avea asociată şi o valoare, un argument al excepţiei. Prezenţa argumentului şi tipul acestuia depinde de tipul excepţiei. Pentru acele excepţii care au un argument, clauza `except` poate avea nume de varibilă, după numele excepţiei, variabilă ce va conţine(dacă acea excepţie este detectată) argumentul asociat excepţiei. Următorul exemplu este lămuritor în acest sens:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'name', x, 'undefined'
...
name spam undefined
```

Dacă o excepţie are un argument, acesta va fi afişat în ultima parte a mesajului ce apare când o excepţie nu este tratată. Într-un bloc `try` nu sunt detectate numai excepţiile generate în cadrul funcţiilor apelate:

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

8.4 Generarea excepţiilor

Instrucţiunea `raise` îi permite programatorului să genereze o anumită excepţie :

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

Primul parametru reprezintă tipul excepţiei, iar al doilea este opţional şi reprezintă un eventual argument.

Dacă se vrea să se stie când o excepţie a fost semnalataă, dar nu se intenţionează tratarea ei, o formă mai simplificataă a instrucţiunii `raise` permite reapariţia excepţiei :

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5 Excepţii definite de utilizator

Un programator îşi poate crea propriile excepţii prin creare unei clase de excepţii noi. În mod obişnuit excepţiile pot fi derivate din clasa `Exception` atât înmod direct cât şi indirect. De exemplu :

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return ‘self.value‘
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print ’My exception occurred, value:’, e.value
...
My exception occurred, value: 4
>>> raise MyError, ’oops!’
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: ’oops!’
```

Clasele de excepţii pot fi proiectate ca oricare alta clasă, dar in mod uzual ele sunt simple, deseori oferind numai un număr de atribute, ce permit să informeze asupra erorii ce a fost tratată. Când se implementează un modul care poate genera un număr distinct de erori, se obişnuieşte să se creeze o clasa de bază a excepţiilor definite de acest modul, şi subclase care să creeze clase specifice pentru diferite condiţii de eroare :

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Excetie generat\u{a} pentru eroare de intrare.

    Atribute:
        expression -- expresia de intrare \^{i}n care apare eroarea
        message -- explicarea erorii
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Generata c\^{a}nd o opera\c{t}ie a\c{s}teapt\u{a} o modificare
    de stare care nu este permis\u{a}

    Attributes:
        previous -- starea la \^{i}nceputul tranzi\c{t}iei
        next -- noua stare care este de a\c{s}tept s\u{a} apar\u{a}
        message -- explica\c{t}ie privind imposibilitatea tranzi\c{t}iei
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Multe excepţii sunt definite cu nume care se termina în “Error,” similar cu denumirea excepţiilor standard.

Multe module îşi definesc propriile excepţii pentru a raporta erorile ce pot apărea înfuncţiile definite de ele. Mai multe informaţii despre clase veţi găsi în capitolul refclasses, “Classes.”.

8.6 Definirea acţiunilor de curăţare

Instrucţiunea `try` poate avea şi o altă clauză prin care se pot defini acţiuni de curăţare care vor fi executate în orice circumstanţă. De exemplu:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print ’Goodbye, world!’
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Clauza *finally* este executată dacă a fost sau nu generată o excepţie în blocul `try`. Când apare o excepţie, este regenerată după ce sunt executate instrucţiunile clauzei `finally`. Clauza `finally` este de asemenea executată “la

ieşire”, chiar şi atunci când a fost apelată o instrucţiune `break` sau `return`.

Codul scris încalauza `finally` este utilă pentru eliberarea resurselor externe (cum ar fi fişiere sau conectări la reţea), fara a mai vedea daca utilizarea resurse s-a încheiat cu succes.

O instrucţiune `try` trebuie fie să aibe una sau mai multe clauze `except`, fie o clauză `finally`, dar nu ambele în acelaşi timp.

CAPITOL

NOUĂ

Clase

Mecanismul de clase al limbajului Python face posibilă adăugarea de noi clase cu un minim de efort. Acest mecanism de clase este o combinaţie între mecanismele din C++ şi Modula-3. La fel ca şi pentru module, Python nu creează o barieră absolută între definiţie şi utilizator, ci se bazează mai mult pe "politeţea" utilizatorului de a nu "pătrunde în definiţie". Cele mai importante facilităţi ale claselor sunt în continuare disponibile.

În terminologie C++, toate componentele unei clase(şi datele membre) sunt publice, şi toate funcţiile membre sunt virtuale. Nu există constructori sau destructori specializaţi. La fel ca în Modula-3, nu există scurtături pentru a accesa membrii unei clase în cadrul unei metode a acelui obiect: metodele sunt definite cu un argument explicit reprezentând obiectul, care este apoi transmis automat în momentul în care funcţia este apelată. Clasele sunt în sine obiecte, iar în sens mai larg: în Python toate tipurile de date sunt obiecte, acest comportament este specific şi limbajului SmallTalk. Prin toate acestea sunt posibile importurile şi redenumirile. La fel ca în C++ şi Modula-3, tipurile predefinite nu pot fi definite drept clase de bază pentru eventuale extensii. De asemenea, la fel ca în C++, dar spre deosebire de Modula-3, majoritatea operatorilor cu sintaxă specială(de exemplu: operatorii aritmetici) pot fi redefiniţi în cadrul claselor.

9.1 Câteva cuvinte despre terminologie

Deoarece nu există o terminologie unanim recunoscută în ceea ce priveşte clasele, nu vom folosi aici termenii din SmallTalk şi C++. Am putea folosi termenii din Modula-3, deoarece semantica acestui limbaj este mai aproape de cea a Python-ului, dar foarte puţini utilizatori au auzit de acest limbaj, deşi C++ şi SmallTalk sunt o alegere mult mai bună, fiind limbaje cu o popularitate mult mai mare.

Va trebui de asemenea să avertizăm cititorii asupra unei greşeli pe care o pot face în interpretarea celor ce urmează: cuvântul "obiect" în Python nu înseamnă neapărat o instanţiere a unei clase. În C++, Modula-3, de asemenea şi în Python, dar nu şi în SmallTalk, există anumite tipuri de date care nu sunt neapărat clase: numerele întregi şi listele nu sunt clase, dar şi alte tipuri de date mai "exotice", cum sunt fişierele nu sunt clase. Toate tipurile de date în Python au aproximativ acelaşi comportament care este mai uşor de explicat dacă ne referim la aceste tipuri de date folosind cuvântul "obiect".

Obiectele pot avea individualitate, sau pot avea mai multe nume(care pot servi mai multor scopuri) care să desemneze acelaşi obiect. Aceast procedeu se numeşte "aliasing" în alte limbaje. De obicei în Python, la prima vedere, nu sunt încurajate astfel de tehnici, care pot fi foarte bine ignorate când este vorba despre tipurile de date nemodificabile(numere, şiruri, perechi). În orice caz, tehnica aceasta, "aliasing", are şi un efect intenţionat asupra sintaxei Python care priveşte obiectele alterabile (liste, dicţionare, etc.) şi obiectele ce există în afara contextului în care se află programul (fişiere, ferestre. etc.). Puteţi utiliza tehnica "aliasing", pentru a beneficia de puterea oferită în alte limbaje, de pointeri, întrucât puteţi considera că alias-urile funcţionează într-un anumit sens ca pointerii din C. Spre exemplu transmiterea unui obiect este mai simplă, din moment ce se transmite doar un pointer. Dacă o funcţie primeşte ca parametru un obiect pe care îl modifică, cel ce a aplelat funcţia va putea observa efectul funcţiei, deoarece modificările efectuate de funcţie au fost făcute asupra conţinutului obiectivului şi nu asupra unui reprezentări locale(în contextul funcţiei). Explicând toate acestea se observă necesitatea a două mecanisme de transmitere a parametrilor ca în Pascal.

9.2 Domenii de vizibilitate (Scopes) şi domenii de definiţie a numelor(Name Spaces)

[1]

Definiţiile claselor manipulează într-un mod foarte interesant domeniile de definiţie a numelor, şi o foarte bună înţelegere a modului de funcţionare a domeniilor de vizibilitate şi de definiţie a numelor este foarte impotantă.

Înţelegerea acestor mecanisme este foarte importantă şi se va dovedi foarte utilă oricărui programator avansat Python.

Să începem cu câteva definiţii:

Un domeniu de definiţii a numelor este o "hartă" de legături între nume şi obiecte. În momentul de faţă aceste domenii sunt implementate folosind dicţionare, dar metodele de implementare pot fi schimbate pe viitor. Exemple de astfel de domenii sunt: domeniul numelor predefinite, domeniul numelor globale dintr-un modul, domeniul numelor locale creat la apelarea unei funţii, etc. Într-un anumit sens atributele unui anumit obiect formează un domeniu de definiţie. Un lucru foarte important de ştiut este acela că între două nume existente în două domenii diferite nu există absolut nici o legătură. Din această cauză două module pot defini o funcţie cu aceeaşi denumire fără a se creea confuzie.

Un atribut este orice nume care urmează unui punct, de exemplu în expresia `z.real`,`real` este un atribut al obiectului `z`. Într-un sens strict, referinţele la nume definite într-un modul sunt de fapt referinţe la atribute ale modulului: în expresia `modul.funcţie`, `modul` este modulul iar `funcţie` este un atribut al modulului.

Numele globale ale modulului şi atributele modulului se află în acelaşi domeniu de definiţie al numelor.(Există o singură excepţie. Modulele au un atribut nemodificabil (read-only) "secret": `__dict__` în care se află domeniul de definiţie al numelor ce aparţine modulului. `__dict__` este un atribut dar nu este un nume global.)

Atributele pot fi nemodificabile(read-only) sau modificabile(writeable). Atributele unui modul sunt modificabile: puteţi folosi o secvenţă de genul: '`modul.răspuns = 42`'. Atributele modificabile pot fi de asemenea şterse folosind operatorul `del`. De exemplu '`del modul.răspuns`' va elimina atributul `răspuns` din obiectul denumit `modul`.

Domeniile de definiţie a numelor sunt create la momente de timp diferite şi au perioade de viaţă diferite. Domeniul care conţine numele predefinite este creat odată cu pornirea interpretorului Python şi nu este şters niciodată. Domeniul numelor globale definite într-un modul este creat atunci când este încărcată definiţia modulului şi este distrus de obicei tot în momentul în care interpretorul este închis. Instrucţiunile executate la primul nivel al interpretorului fie dintr-un script, fie interactiv, fac parte dintr-un modul numit `__main__`. Numele predefinite se află în domeniul `__builtin__`.

Domeniul de definiţie a numelor pentru o funcţie este creat odată cu apelul funcţiei, şi este şters odată cu terminarea funcţiei, fie prin `return`, fie dacă acesta generează o excepţie netratată. Bineînţeles că apeluri recursive generează mai multe domenii de definiţie.

Un domeniu de vizibilitate (scope) este o regiune a unui program Python în care un anumit domeniu de definiţie a numelor este accesibil. Accesibil înseamnă că o anumită referinţă la un nume va genera o căutare în domeniul de definiţie al acelui modul dspre care spunem că este accesibil.

Domeniile de vizibilitate sunt determinate static, şi sunt folosite dinamic. În timpul execuţiei există exact trei domenii de vizibilitate (exact trei domenii de definiţie sunt direct accesibile): primul "cel mai adânc", conţine numele locale, cel de-al doilea conţine numele globale ale modulului curent, iar cel de-al treilea "cel mai de sus", care este ultimul în care se caută un nume(dacă în celelalte căutarea nu a avut succes, şi care conţine numele predefinite.

De obicei, "cel mai adânc" domeniu de vizibilitate este cel al funcţiei care este executată. În exteriorul unei funcţii, acest domeniu este identic cu al doilea domeniu (cel din mijloc), în care se află accesibile definiţiile modulului.

Domeniul de vizibilitate global al unei funcţii definite într-un modul este domeniul de definiţii al respectivului modul, indiferent de unde, sau cu ce alias este apelată respectiva funcţie. Pe de altă parte căutarea unui anumit nume este realizată dinamic, în timpul execuţiei. Cu toate astea definiţia limbajului evoluează către o "rezolutie" a numelor

[1](N.T. Traducerile celor doi termeni în limba română sunt improprii, iar înţelegerea acestor termeni ar trebui să fie mai mult o înţelegere "contextuală" nedând o prea mare importanţă sensului pe care o are traducerea celor doi termeni).

statică, în timpul compilării. În concluzie nu vă bazaţi prea tare pe varianta dinamică. Deja variabilele locale sunt determinate static.

O caracteristică a limbajului Python, aste că atribuirile au loc în primul domeniu de vizibilitate("cel mai de jos" sau "cel mai adânc" N.T. termenul original este `innermost`). Atribuirile nu realizează o copiere a datelor ci leagă nume de obiecte. Acest lucru este valabil şi pentru ştergeri. Instrucţiunea `del x` şterge legătura lui x din domeniul local de definiţie. De fapt toate operaţiile care introduc nume noi au impact asupra domeniului de definiţii local: instrucţiunile `import` sau definiţia unei funcţii introduc numele modulului importat, respectiv numele funcţiei în domeniul de definiţii local. Dacă este folosită instrucţuinea `global` se indică intrepretorului ca următorul nume să fie introdus în domeniul global.

9.3 O primă privire asupra claselor

Noţiunea de clase aduce cu sine şi câteva alte noţiuni noi de sintaxă, trei tipuri noi de obiecte, şi câteva noţiuni noi de semantică.

9.3.1 Definirea unei clase

Cea mai simplă definiţie a unei clase arată astfel:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Definiţiile de clase, la fel ca definiţiile de funcţii, sunt valabile de abia după ce sunt executate. Puteţi insera o definiţie de clasă în cadrul unei instrucţiuni `if`, şi clasa nu va exista decât dacă condiţia instrucţiunii `if` va fi adevărată.

În practică, instrucţiunile din interiorul unei clase vor fi definiţii de funcţii, dar sunt permise şi alte instrucţuni care pot fi foarte utile uneori. Definiţiile de funcţii din cadrul unei clase sunt un pic deosebite faţă de ce cunoaşteţi, dar despre asta vom vorbi mai târziu.

Atunci când este introdusă definiţia unei clase noi, un nou domeniu de definiţii a numelor este creat şi folosit ca domeniu de vizibilitate local, deci toate atribuirile vor afecta noul domeniu de definiţii creat. Definiţiile de funcţii vor introduce numele acelor funcţii tot în acest domeniu.

Când definiţia unei clase se termină normal, fără erori, este creat un obiect nou de tipul `class`. Pe scurt, acest obiect "conţine" noul domeniu de definiţii creat odată cu definiţia clasei. În următoarea secţiune vom discuta mai mult despre obiecte de tip `class`. Domeniul de vizibilitate existent înaintea definiţiei clasei este reactivat, şi obiectul `class` este introdus în respectivul domeniu de definiţii sub numele dat de numele specificat în definiţia clasei (în exemplul de mai sus: `ClassName`).

9.3.2 Obiecte class

Această categorie de obiecte acceptă două categorii de operaţii: referenţieri de atribute şi instanţieri.

Referenţierile de atribute folosesc sintaxa standard din Python: `obiect.atribut`. Atribute valide sunt toate numele existente în domeniul de definiţii în momentul creării obiectului `class`. Aşa că, dacă definiţia clasei arată astfel:

```
class MyClass:
    "Un exemplu simplu de clasa"
    i = 12345
    def f(self):
        return 'hello world'
```

Atunci `MyClass.i` şi `MyClass.f` sunt referinţe la atribute valabile. Unul dintre atribute este un întreg (i), iar celălalt o metodă(f). Se pot realiza atribuiri cu atributele unei clase, deci puteţi modifica valoarea atributului `MyClass.i` printr-o atribuire. `__doc__` este de asemenea un atribut al clasei, care are valoare "Un exemplu simplu de clasa".

Instanţierea claselor foloseşte notaţia de la funcţii. Putem să ne imaginăm că obiectul clasă este o funcţie fără parametrii care întoarce ca rezultat o nouă instanţă pentru respectiva clasă: Exemplul următor:

```
x = MyClass()
```

creează o nouă instanţă a clasei şi atribuie acest obiect variabilei locale x.

Operaţia de instanţiere crează un obiect vid. Multe clase preferă să creeze un obiect într-o stare iniţială. Pentru a realiza acest lucru o clasă trebuie să aibă definită o metodă specială numită `__init__()` ca în exemplul de mai jos:

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

Atunci când o clasă are definită metoda `__init__()`, instanţierea respectivei clase apelează automat metoda `__init__()`. În acest fel se poate obţine o instanţă iniţializată a unei anumite clase:

```
x = MyClass()
```

Bineînţeles metoda `__init__()` poate avea argumente. În acest caz argumentele primite la instanţierea clasei sunt transmise automat metodei `__init__()`. De exemplu:

```
>>> 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)
```

Vă puteţi gândi la metoda `__init__()` ca la un constructor, deşi nu este un constructor în toată puterea cuvântului.

9.3.3 Obiecte instanţiate

Ce putem face cu instanţele? Singurele operaţii acceptate sunt operaţiile cu atribute. Există două tipuri de nume de atribute valide.

Prima categorie: propietăţile sau atribute de tip dată. Această categorie corespunde "variabilelor instanţei" în SmallTalk, şi "membrilor dată" în C++, create automat atunci când le sunt atribuite valori. Dacă x este o instanţă a clasei `MyClass` următoarea porţiune de cod va afişa valoarea 16:

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

A doua categorie sunt `metodele`. O metodă este o funcţie ce aparţine unui anumit obiect.(În Python termenul de metodă nu desemnează doar o funcţie ce aparţine instanţei numai clase, adică unui obiect; şi alte tipuri de obiecte au metode. De exemplu, obiectele listă au metodele: `append`, `insert`, `sort`, etc. În cele ce urmează vom folosi termenul de metodă numai în sensul mai sus definit).

Numele de metode disponibile pentru un anumit obiect depind de clasa din care face parte obiectul. Prin definiţie, toate atributele de tip funcţie ale unei clase devin metode pentru toate instanţele respectivei clase. În exemplul nostru `x.f` este o metodă valabilă din moment ce `MyClass.f` este o funcţie, în mod contrar, este evident că `x.i` nu este o metodă. `MyClass.i` nu este o funcţie. Există o diferenţă între `MyClass.f` şi `x.f`: prima este un obiect funcţie iar cea de-a doua un obiect metodă(una este funcţie, a doua metodă).

9.3.4 Obiecte metodă

În mod uzual o metodă este apelată imediat:

```
x.f()
```

În exemplul nostru aceasta va returna şirul de caractere `Hello World!`. Oricum nu este neapărat necesar să apelăm o metodă. Putem face acest lucru mai târziu:

```
xf = x.f
while 1:
        print xf()
```

Exemplul de mai sus va apela `Hello World!` la infinit.

Ce se întâmplă exact atunci când este apelată o metodă? Aţi observat că `x.f` a fost apelată fără nici un argument, chiar dacă în definiţia funcţiei era specificat un argument.

Ce s-a întâmplat cu argumentul? Cu siguranţă Python generează o excepţie atunci când o funcţie care necesită un argument(deşi poate să nu aibe neapărat nevoie de acesta), este apelată fără argumente.

Probabil că aţi ghicit deja răspunsul: metodele au întotdeauna ca prim parametru un obiect.În exemplul nostru apelul `x.f()` este absolut echivalent cu `MyClass.f(x)`. În general a apela o metodă cu n parametrii este echivalent cu a apela funcţia corespondentă inserând la începutul listei de n argumente obiectul a cărui metodă se doreşte apelată.

Dacă nu aţi înţeles încă cum funcţionează metodele, o privire asupra implementării probabil că ar clarifica lucrurile. Atunci când se face o referire la un atribut al unei instanţe care nu este un atribut dată, clasa din care face parte este căutată. Dacă numele denotă un atribut existent în respectiva clasă(deci un obiect de tip funcţie), este creat un obiect metodă punând laolaltă(“împachetând”) obiectul instanţă şi obiectul funcţie(tocmai găsit). În acest fel se creeaza un obiect metodă. Atunci când o metodă este apelată cu o listă de argumente, obiectul metodă este despachetat, o nouă listă de argumente este construită adăugând la începutul listei de argumente obiectul instanţă ca prim parametru. În cele din urmă obiectul funcţie este apelat cu noua listă de argumente.

9.4 Alte observaţii

Atributele dată suprascriu atributele metodă cu acelaşi nume. Pentru a evita conflicte de nume ce pot cauza defecţiuni greu de găsit în programe mari, este bine să folosiţi o convenţie pentru denumiri pentru a minimiza şansa apariţiei unor astfel de conflicte. O posibilă convenţie ar fi scrierea numelelor de metode cu literă iniţială mare, prefixarea numelor de atribute dată cu un şir de caractere mic şi unic, (poate fi un underscore), sau pentru metode folosiţi verbe, iar pentru propietăţi substantive.

Propietăţile pot fi accesate atât de metode, cât şi de utilizatorii unui anumit obiect. Cu alte cuvinte clasele nu pot fi folosite pentru a implementa tipuri de date abstracte. De fapt, în Python nu există nici un fel de mecanism de ascundere a datelor – aceasta realizându-se numai prin convenţie. (Pe de altă parte, implementarea limbajului Python, relizată în C, poate ascunde complet detalii de implementare controlând accesul la datele unui obiect, dacă o astfel de abordare este necesară).

Utilizatorii unui obiect ar trebui să manipuleze atributele dată (propietaţile) cu atenţie – orice modificare directă a acestora putând duce la o "incoerenţă" a obiectului respectiv. De notat este faptul că utilizatorii unui obiect, pot adăuga propriile atribute dată fără a afecta în vreun fel metodele, atâta timp cât nu apar conflicte de nume(şi aici existenţa unei convenţii vă poate scuti de multe neplăceri).

Nu există nici o scurtătură pentru referirea atributelor din interiorul unei metode. Acest lucru clarifică codul unei metode, nemaiexistând posibilitatea confuziei între variabilele locale şi variabilele instanţei.

În mod convenţional, primul argument al unei metode are numele: `self`. Aceasta este numai o convenţie, numele `self` neavând nici o semnificaţie specială în Python.(Folosind această convenţie codul dumneavoastră va fi mai uşor de citit de către programatori).

Orice obiect funcţie ca atribut al unei clase defineşte o metodă asociată pentru instanţele unei clase. Nu este neapărat necesar ca definiţia unei funcţii să se afle în cadrul definiţiei clasei. În acest sens se poate atribui un obiect funcţie unei variabile locale. De exemplu:

```
# Func\c{t}ie definita \^{i}nafara clasei

def f1(self, x, y):
    return min(x, x+y)


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

Acum `f`,`g`, şi `h` sunt toate atribute ale clasei C, şi sunt atribute funcţiei, şi în mod consecvent sunt metode pentru toate instanţele clasei C. Această practică nu este indicată, întrucât nu face decât să încurce cititorul programului.

Metodele pot apela alte metode utilizând metoda atributelor argumentului `self` :

```
class Bag:
    def empty(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Metodele pot face referiri la nume globale în acelaşi mod ca funcţiile obişnuite. Domeniul global de vizibilitate asociat unei metode permite accesul la domeniul de definiţie a numelor asociat modului în care se află definită clasa căreia îi aparţine metoda.

Clasa în sine nu este folosită niciodată ca un domeniu global de vizibilitate. Pot fi găsite multe utilizări ale domeniului global de vizibilitate, deşi unii ar putea considera inutilă existenţa acestuia.

9.5 Moştenirea

Nu putem spune despre un limbaj că este orientat obiect, dacă sistemul de clase nu pune la dispoziţie mecanismul de moştenire. Sintaxa pentru moştenire (adică pentru crearea definiţiei unei clase derivate dintr-o clasă de bază) este următoarea:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Clasa `BaseClassName` trebuie să fie definită pentru ca definiţia clasei derivate să fie corectă. În loc de un nume de clasă puteţi folosi o expresie, acest lucru fiind foarte util atunci când clasa de bază se află într-un alt modul:

```
class DerivedClassName(modname.BaseClassName):
```

Atunci când obiectul `class` pentru clasa derivată, este creat, este memorată şi clasa de bază. Acest mecanism este folosit pentru a rezolva referirile la atribute: în caz că un atribut nu este găsit în clasa derivată, este căutat şi în clasa de bază. Regula se aplică recursiv dacă clasa de bază este la rândul ei o clasă derivată.

Instanţierea unei clase se face la fel ca până acum. Dacă este apelată o metodă a unei instanţe, numele metodei este căutat întâi în definiţia clasei derivate, apoi în definiţia clasei de bază, şi aşa mai departe până când este găsit un obiect funcţie corespunzător în lanţul de moşteniri.

Clasele derivate pot suprascrie metode ale clasei de bază. Deoarece o metodă nu dispune de nici un privilegiu atunci când apelează o altă metodă a aceluiaşi obiect, o metodă a clasei de bază care apelează o altă metodă, definită de asemenea în clasa de bază, poate să apeleze de fapt o metodă suprascrisă de clasa derivată (pentru programatorii C++ putem spune că toate metodele din Python sunt virtuale).

O astfel de metodă suprascrisă poate chiar să extindă metoda de bază, în loc de a o înlocui. Există o cale simplă de a apela o metodă a clasei de bază direct: `ClasădeBază.metodă(self.argumente)`.

Acest lucru poate fi foarte util chiar şi utilizatorilor(clienţilor) unei clase.

9.5.1 Moştenirea multiplă

Python pune la dispoziţia programatorilor şi o formă de moştenire multiplă. O definiţie de clasă derivată din mai multe clase de bază arată în felul următor:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Regula după care sunt căutate atributele este următoarea: dacă un atribut nu este definit în clasa derivată atunci acesta este căutat în definiţia clasei `Base1` şi în toate clasele de la care aceasta moşteneşte, apoi este căutat în `Base2` şi în `Base3` după aceleaşi reguli. Putem rezuma această regulă în felul următor: "întâi în adâncime, şi de la stânga la dreapta". (Avantajele şi dezavantajele acestei reguli pot fi discutate, însă trebuie să fie foarte clar pentru toată lumea că o astfel de regulă este absolut necesară, fapt suficient de evident).

Este suficient de clar că folosirea mecanismului de moştenire multiplă va crea mari probleme de întreţinere, deoarece în Python conflictele ce pot apărea între nume sunt evitate numai prin convenţii. O problemă cunoscută apare atunci când o clasă moşteneşte de la două clase care la rândul lor sunt derivate dintr-o singură clasă. Deşi este destul de simplu să ne imaginăm ce se va întâmpla în acest caz, nu este clar că regulile vor fi de folos pentru clarificarea situaţiei.

9.6 Variabile private

Există un suport limitat pentru identificatorii clselor private. Orice identificator de forma `__spam` (cel puţin două underscore înainte, cel mult un underscore după) este acum textual înlocuit cu `_clasă__spam`, unde `clasă` este numele curent al clasei cu underscor-urile din faţă suprimate. Mutilarea este realizată fara o analiza a poziţiei sintactice a identificatorului, aşa încât putând fi folosită la definirea unei instanţieri a unei clase private şi a variabilelor şi metodelor clasei ca globale, şi la memorarea instanţierii variabilelor private înaceastă clasă la instanţierea *altei* clase. O trunchere a numelui poate apărea când acesta este mai lung de 255 de caractere. În afara claselor, sau când numele clasei este alcătuit numai din underscor-uri nu se face eliminarea underscor_urilor.

Acest mecanism pune la dispoziţia programatorului o metodă foarte facilă de a crea atribute private, fără a-şi face griji că aceste atribute pot provoca conflicte de nume. Toate acestea sunt realizate în Python pentru a preveni eventualele erori, totuşi, este în continuare posibilă modificarea atributelor private, ceea ce se poate dovedi foarte util în anumite cazuri ca de exemplu la depanare. (Observatie : o clasă derivată care are acelaşi nume cu clasa de bază poate folosi variabilele private ale clasei de bază).

Observaţi că, codul ce este trimis instrucţiunilor `exec, eval()` sau `evalfile()` nu ia în considerare numele clasei care realizează invocările, acest efect fiind similar celui realizat prin instrucţiuni `global`, efect ce nu poate fi obţinut pentru cod compilat. Aceeaşi restricţie se aplică şi pentru `getattr(), setattr()` şi `delattr()`, de asemenea şi pentru accesarea directă a dicţionarului `__dict__`.

Iată un exemplu de clase care implementează propriile metode `__getattr__()` şi `__setattr__()`, şi stochează toate datele în variabile private:

```
class VirtualAttributes:
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7 Altfel de clase

Câteodată, poate fi foarte util să putem crea tipuri de date similare celei `record` din Pascal, sau `struct` din C, tipuri de date care să adune la un loc mai multe date de diferite tipuri. Pentru a realiza acest lucru puteţi folosi clasele. O definire a unei clase vide se poate face astfel :

```
class Employee:
    pass


john = Employee() # Se creaz\u{a} o \^{i}nregistrare vid\u{a} de tip employee


# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Să construim un alt scenariu: presupunând că aveţi o funcţie care prelucrează anumite date dintr-un obiect fişier, puteţi să definiţi o clasă cu metodele `read()` şi `readline()` care să citească informaţii dintr-un şir de caractere în loc de un fişier.

Metodele obiectelor au de asemenea atribute: `metodă.im__self` reprezintă obiectul de care aparţine metoda specificată, iar `metodă.im__func` reprezintă funcţia corespunzătoare metodei.

9.7.1 Excepţiile pot fi clase

Excepţiile definite de utilizator pot fi atât obiecte de tip `string`, cât şi clase. Folosind clasele pentru a defini excepţii, puteţi crea o întreagă ierarhie de excepţii care poate fi foarte uşor extinsă.

Există o altă formă a insrucţiunii `raise`:

```
raise Class, instance

raise instance
```

În prima formă `instance` trebuie să fie o instanţă a clasei `Class` sau a unei clase derivate. A doua formă este o prescurtare pentru:

```
raise instance.__class__, instance
```

O clauză `except` poate afişa atât excepţii de tip clasă, cât şi de tip şir de caractere. Următorul exemplu va afişa B, C, D în această ordine:

```
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"
```

De observat că dacă ordinea clauzelor `except` ar fi fost inversă, (cu `except B` la început), exemplul ar fi afişat B, B, B; deoarece clasa B este clasa din care sunt derivate clasele C şi D.

Când un mesaj de eroare este afişat pentru o excepţie netratată ce provine dintr-o clasă, numele clasei este de asemenea afişat.

CAPITOL

ZECE

Continuarea?

Să sperăm că acest tutorial v-a trezit interesul în ceea ce priveşte limbajul Python. Ce ar trebui să faceţi de acum încolo?

Ar trebui să citiţi, sau cel puţin să răsfoiţi *Python Library Reference*,document ce explică foarte detaliat fiecare funcţie, tip de date, modul, etc. existent în Python. Distribuţia standard de Python conţine o cantitate imensă de cod scris atât în C, cât şi în Python, există module UNIX ce pot fi folosite pentru a citi căsuţele de poştă electronică, module care facilitează lucrul cu protocolul HTTP, module pentru generarea numerelor aleatoare, etc.

Un site foarte util este `http://www.python.org`. Aici există documentaţii bucăţi de cod scrise de alţi programatori şi link-uri legate de Python.

Copii ale acestui site există pe serverele din întreaga lume (Europa, Australia, Japonia, etc). În funcţie de locul în care vă aflaţi, pentru o viteză de acces mai mare, puteţi folosi una dintre aceste copii. Un alt site ce se poate dovedi foarte util este `http://starship.python.net`.

Pentru întrebări sau raportarea anumitor probleme puteţi folosi grupul comp.lang.python, sau puteţi scrie la Python-list@python.org. În fiecare zi sunt puse în jur de 120 de întrebări care, de obicei, îşi găsesc răspunsul.

Înainte de a pune o întrebare, verificaţi lista întrebărilor puse frecvent (Freqvently Asked Questions = FAQ) care poate fi găsită la adresa `http://www.python.org/doc/faq.html`. Răspunsurile de la această adresă, la diferite întrebări puse de alţi programatori, vă pot ajuta de foarte multe ori în rezolvarea problemelor dumneavoastră.

ANEXA

A

Editarea în linie de comandă şi repetarea comenzilor anterioare

Unele versiuni ale interpretorului Python suportă editarea în linie de comandă şi repetarea unor comenzi anterioare, facilităţi întâlnite în shell-urile Korn sau GNU Bash. Aceste facilităţi sunt implementate folosind biblioteca **GNU Readline** care suportă stilurile de editare specifice `Emacs` şi `Vi`. Această bibliotecă are propria documentaţie pe care nu o vom reproduce aici, ci vom face numai o scurtă introducere.

Facilităţile despre care vă vom vorbi sunt disponibile în versiunile UNIX şi CygWin ale interpretorului.

Acest capitol nu va explica facilităţile pachetelor Mark Hammond's Python Win sau IDLE distribuite odată cu Python.

A.1 Editarea în linie de comandă

Dacă este suportată, această facilitate va fi disponibilă atunci când interpretorul afişează promptul principal sau cel secundar.

Linia curentă poate fi editată folosind caracterele de control convenţionale Emacs.

Cele mai importante sunt:

- **C-A**(Control-A) mută cursorul la începutul liniei;
- **C-E**(Control-E) mută cursorul la sfârşitul liniei;
- **C-B** mută cursorul cu o poziţie la stânga;
- **C-F** mută cursorul cu o poziţie la dreapta;
- **BACKSPACE** şterge caracterul de la stânga cursorului;
- **C-D** şterge caracterul de la dreapta cursorului;
- **C-K** şterge tot ce urmează pe linie în dreapta cursorului;
- **C-Y** reface ultimul şir şters cu **C-K**;
- **C-_** anulează ultima comandă efectuată. Operaţia poate fi repetată.

A.2 Repetarea comenzilor anterioare(History)

Toate comenzile executate sunt salvate într-o zonă tampon (Buffer). Atunci când este executată o nouă instrucţiune aceasta este salvată la sfârşitul buffer-ului.

C-P revine la o comandă anterioară în buffer celei executate.

C-M înaintează la următoarea instrucţiune din buffer.

Orice linie din buffer poate fi modificată. Dacă o linie a fost modificată veţi vedea un caracter '*' în faţa acesteia care îndică că a fost modificată. O instrucţiune din buffer este executată dacă se apasă, evident, **ENTER**.

C-R porneşte o căutare incrementală în buffer în sens invers.

C-S porneşte o căutare în sens normal.

A.3 Redefinirea tastelor funcţionale

Tastele funcţionale şi alţi parametrii ai bibliotecii "Readline", pot fi redefiniţi prin modificarea fişierului "~/.inputrc". Pentru a redefini funcţia unei combinaţii de taste trebuie folosită sintaxa:

`tastă:funcţie` sau `"şir de caractere":funcţie`.

O opţiune poate fi setată folosind

`set opţiune valoarenouă`.

Iată un exemplu:

```
# Prefer stil de editare Vi:
set editing-mode vi
# Editare pe o singura linie:
set horizontal-scroll-mode On
# Redefinesc cateva taste:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

În Python principala funcţie pentru `Tab` este aceea de a insera `Tab`, nu funcţia de completare Readline. Dacă se doreşte însă utilizarea sa în funcţia de completare se va pune :

```
Tab: complete
```

în '~/.inputrc'. (Bine înţeles că acum va fi mai dificil de a scrie spaţiile de aliniere)

Completarea automată a numelui unei variabile, sau a unui modul, este opţională. Pentru a o activa în modul de lucru interactiv adăugaţ în fişierul de startup următoarele :[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Aceasta leagă tasta `Tab` de funcţia de completare, deci apăsând tasta `Tab` de doua ori activează completarea; analizează numele instrucţiunii Python, variabilele locale curente şi numele de module disponibile.Pentru expresiile cu punct de tipul `string.a`, evaluează expresia până la final, după `'.'` şi apoi sugerează completarea dată de atributele obiectelor rezultate. De reţinut că se poate executa codul unei aplicaţii definite dacă un obiect ce are metoda `__getattr__()` face parte din expresie.

Un fişier startup mai cuprinzător poate arăta ca în exemplul următor. De notat că el işi sterge numele pe care le crează în momentul în care nu-i mai sunt necesare. Acest lucru se petrece când fişierul de starup se execută în acelaşi domeniu de definiţie ca şi comenzile interactive, iar eliminarea numelor evită generarea unor efecte colaterale în mediul interactiv.

[1] Python va executa conţinutul unui fişier identificat prin variabila de mediu PYTHONSTARTUP în momentul pornirii interpretorului interactiv.

Puteţi găsi acest lucru convenabil pentru a păstra câteva module importate, cum ar fi `os`, şi care se dovedesc necesare în majoritatea sesiunilor interpretorului.

```
# La interpretorul interactiv Python se adaug\u{a} func\c{t}ia de autocompletare
# \c{s}i un fi\c{s}ier de tip jurnal pentru comenzi. Se cere Python 2.0+,
# readline. Autocompletarea este executat\u{a} implicit prin tasta Esc (dar,
# tasta poate fi \^{i}nlocuit\u{a} -- vezi documenta\c{t}ia readline).
#
# Pune\c{t}i fi\c{s}ierul \^{i}n ~/.pystartup, iar variabila de mediu s\u{a}-l
# indice, cum ar fi \^{i}n bash "export PYTHONSTARTUP=/max/home/itamar/.pystartup".
#
# Re\c{t}ine\c{t}i c\u{a} PYTHONSTARTUP nu expandeaz\u{a} "~", deci va trebui
# indicat\u{a} \^{i}ntraga cale a directorului home .

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 Comentarii

Aceste facilităţi reprezintă un pas enorm comparativ cu versiunile anterioare ale interpretorului; oricum au rămas câteva deziderate neâmplinite : ar fi frumos ca să fie sugerat un alineat corect la continuarea unei linii (analizorul sintactic ştie dacâ se va cere un alineat). Acest nou mecanism ar putea utiliza tabela de simboluri a interpretorului. Util ar fi şi un mecanism care să verifice (sau să sugereze) potrivirea parantezelor, ghilimelelor, etc.

ANEXA

B

Aritmetica în virgulă flotantă: rezultate şi limitări

Numerele în virgulă flotantă sunt reprezentate în hardware-ul calculatorului ca fracţii în baza 2 (binare). De exemplu, fracţia zecimală

```
0.125
```

are valoarea 1/10 + 2/100 + 5/1000, şi în acelaşi fel fracţia binară

```
0.001
```

are valoarea 0/2 + 0/4 + 1/8. Aceste două fracţii au aceeaşi valoare, singura diferenţă reală fiind că prima este scrisă în baza 10, iar a doua în baza 2. Din păcate majoritatea fracţiilor zecimale nu pot fi reprezentate corect ca fracţii binare. O primă concluzie este că, (în general, numerele în virgulă flotantă zecimală pe care le introduceţi au numai o reprezentare aproximativă prin numerele în virgulă flotantă binară memorate în calculator.

Problema este uşor de înţeles întîi în baza 10. Să considerăm fracţia 1/3. O puteţ aproxima cu o fracţie î baza 10:

```
0.3
```

sau, mai precis,

```
0.33
```

sau ,şi mai precis,

```
0.333
```

şi tot aşa. Nu contează ce număr de digiţi veţi scrie, rezultatul nu va fi nicicând exact 1/3, dar creşte acurateţea aproximării lui 1/3.

În acelaşi mod, nu contează câti digiţi în baza 2 folosiţi, valoarea zecimală 0.1 nu va fi reprezentată exact niciodată printr-o fracţie. În baza 2, 1/10 este o fracţie care se repetă la infinit

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

Oprirea la un număr finit de biţi dă o aproximare. Iată ce se afişează în situaţia următoare :

```
>>> 0.1
0.10000000000000001
```

Pe majoritatea maşinilor existente astăzi acest rezultat se afişează dacă introduceţi 0.1 la prompterului Python-ului. Se întâmplă aşa pentru că numărul de biţi utilizaţi de hardware să mememoreze valoarea în virgulă flotantă variază de la o maşină la alta, iar Python listează doar o aproximaţie zecimală a valorii zecimale reale a aproximaţiei binare memorată în maşină. Pe majoritatea maşinilor, dacă Python va tipări valoarea zecimală reală a aproximaţiei binare memorate pentru 0.1, va afişa

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Prompterul Python utilizează funcţia implicită `repr()` ca să obţină o versiune de şir pentru toate obiectele pe care le afişează. În cazul numerele flotante `repr(`*float*`)` rotunjeşte valoarea zecimală la 17 digiţi semnificativi, astfel

```
0.10000000000000001
```

`repr(`*float*`)` foloseşte 17 biţi semnificativi pentru că sunt consideraţi suficienţi pe majoritatea maşinilor, aşa că relaţia `eval(repr(`*x*`)) ==` *x* este exactă pentru toate numerele flotante finite *x*, dar rotunjirea la 16 biţi nu este suficientă pentru a face relaţia adevărată.

Acesta este un lucru normal al aritmeticii binare în virgulă flotantă : nu este un bug Python, nici un bug al codului dumneavoastră şi veţi constata o astfel de anomalie în toate limbajele care folosesc aritmetica în virgulă flotantă oferită de hardware (unele limbaje pot să nu *afişeze* implicit diferenţa)

Funcţia internă Python `str()` generează numai 12 digiţi semnificativi şi poate că doriţi să o folosiţi modificată. Este des utilizată funcţia `eval(str(`*x*`))` pentru a-l obţine pe *x*, dar este mai placut să se afişeze :

```
>>> print str(0.1)
0.1
```

E bine de ştiut că, în sens real, este o iluzie : în maşină valoarea nu este exact 1/10, valoarea adevărată din maşină este rotunjită la *afişare*.

Această surpriză este urmata de altele. De exemplu după ce se afişează

```
>>> 0.1
0.10000000000000001
```

apare tentaţia să utilizaţi funcţia `round()` ca să trunchiaţi iarăşi la un digit. Dar aceasta nu face nimic:

```
>>> round(0.1, 1)
0.10000000000000001
```

Problema este că valoarea memorată învirgulă flotantă binară pentru "0.1" a fost făcută în cea mai bună aproximare binară posibilă la 1/10, deci orice altă rotunjire ulterioară nu o poate face mai bine : a fost dată deja cea mai bună soluţie.

Altă consecinţă a faptului că 0.1 nu este exact 1/10 este aceea că adăugând la 0.1 pe el însuşi de 10 ori nu se obţine 1.0:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
fs>>> sum
0.99999999999999989
```

Aritmetica binară în virgulă flotantă are multe surprize de acest gen. Problema cu "0.1" este explicată în secţiunea următoare, "Erori de reprezentare". Vezi : *The Perils of Floating Point* privind tratarea completă a altor surprize de acest gen.

Cum s-ar spune la sfârşit: "nu este uşor de răspuns". Totuşi nu disperaţi din cauza firgulei flotante! Erorile operaţiilor de flotantă din Python sunt moştenite de la virgula flotantă a hardware-ului şi la majoritatea maşinilor sunt de un ordin mai mic de 1/2**53 pe operaţie. Acestea sunt mai mult decât acceptabile pentru cele mai multe aplicaţii, dar trebuie să reţineţi că nu sunt datorate aritmeticii zecimale şi că la fiecare operaţie învirgulă flotantă apare o eroare de rotunjire.

Până la apariţia unor cazuri patologice, pentru cele mai multe situaţii de utilizare a aritmeticii învirgulă flotantă se obţin rezultatele aşteptate, dacă pur şi simplu se rotunjeşte rezultatul final afişat la numărul de zecimale dorit. Uzual funcţia `str()` este suficientă, iar pentru un control mai riguros a se vedea discuţia despre operatorul de format `%` : formatele `%g`, `%f` şi `%e` sunt moduri flexibile şi uşoare pentru afişarea rezultatelor.

B.1 Erori de reprezentare

Secţiunea explică îndetaliu exemplul "0.1" şi arată cum se poate face o analiză exactă a cazurilor. În principal se urmăreşte familiarizarea cu reprezentarea binară în virgulă floantă.

Eroarea de reprezentare se referă la faptul că fracţiile zecimale nu se pot reprezenta exact ca fracţii binare (în baza 2). Acesta este motivul major pentru care Python (sau Perl, C, C++,Java, Fortran, şi multe altele) nu afişează valoarea exactă a numărului zecimal pe care-l aşteptaţi :

```
>>> 0.1
0.10000000000000001
```

Ce este asta? 1/10 nu este reprezentat exact ca o fracţie binară. Cele mai multe maşini utilizează astăzi (noiembrie 2000) aritmetica de virgulă flotantă IEEE-754 şi cele mai multe platforme mapează flotantele Python în"dublă precizie" IEEE-754. Dublele 754 au 53 biţi de precizie. Aşa că la intrarea în calculator se chinuie să convertească 0.1 încea mai apropiată fracţie posibilă de forma *J*/2***N*, unde *J* este un întreg de exact 53 de biţi. Se rescrie

```
 1 / 10 ~= J / (2**N)
```

ca

```
J ~= 2**N / 10
```

şi reapelând *J* are exact 53 de biţi (este `>= 2**52` dar `< 2**53`), şi cea mai bună valoare pentru *N* este 56:

```
>>> 2L**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

Asta este, 56 este singura valoare pentru *N* care lasă *J* cu exact 53 de biţi. Atunci cea mai bună valoare posibilă pentru *J* este câtul rotunjit:

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

După ce restul este mai mare decît jumătatea lui 10, cea mai bună aproximaţie se obţine prin rotunjirea:

```
>>> q+1
7205759403792794L
```

Astfel cea mai bună aproximaţie a lui 1/10, în dubla precizie 754, este peste 2**56, sau

```
7205759403792794 / 72057594037927936
```

De remarcat că până la rotunjire, valoarea este de fapt un pic mai mare decât 1/10; dacă nu am rotunji câtul va fi un pic mai mic decît 1/10. Dar în nici un caz nu poate fi *exact* 1/10!

Deci, niciodata computerul nu va “arăta” 1/10: ce va arăta este exact fracţia dată mai jos, cea mai bună aproximaţie în 754 care poate fi dată:

```
>>> .1 * 2L**56
7205759403792794.0
```

Dacă se înmulţeşte fracţia cu 10**30, vom vedea valoarea (truncheată) a celor mai semnificativi 30 de digiţi zecimali ai săi:

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000005551115123125L
```

însemnând că numărul exact memorat în calculator este aproximativ egal cu valoarea zecimală 0.100000000000000005551115123125. Rotunjind la 17 digiţi semnificativi rezultă 0.10000000000000001 pe care Python îl afişează (va afişa pe orice platformă conformă 754 care va face tot posibilul să convertească intrare şi ieşire prin biblioteca C proprie — al dumneavoastră s-ar putea să nu reuşească acest lucru!)

ANEXA

C

Istoria şi licenţa

C.1 Istoricul produsului Python

Python a fost creat la începutul anilor 1990 de către Guido van Rossum la Stichting Mathematisch Centrum (CWI, vezi http://www.cwi.nl/) din Olanda, ca succesor al limbajului ABC. Guido a rămas principalul autor al Python-ului, care acumulează multe contribuţii ale altora.

în 1995 Guido işi continuă munca la Python la Corporation for National Research Initiatives (CNRI, vezi http://www.cnri.reston.va.us/) în Reston, Virginia, unde a lansat câteva versiuni ale software-ului.

în mai 2000, Guido şi echipa de dezvoltare a Python-ului s-a mutat la BeOpen.com, unde a format echipa BeOpen PythonLabs. î octombrie 2000, colectivul PythonLabs s-a mutat la Digital Creations (vezi http://www.digicool.com/).în 2001 ia naştere Python Software Foundation (PSF, vezi http://www.python.org/psf/) o organizaţie non profit creată special pentru a deţne drepturile de autor[1] referitoare la Python. Digital Creations este unul din sponsorii PSF.

Toate lansările[2] Python sunt Open Source (vezi http://www.opensource.org/ pentru a găsi definiţia Open Source). Istoriceşte vorbind cele mai multe dintre lansările Python, nu toate, sunt de asemenea compatibile GPL(General Public License). Tabelul următor face un sumar al diferitelor lansări.

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

Notă: Compatibil GPL nu înseamnă că Python este distribuit sub GPL. Toate licenţele Python, spre deosebire de GPL, iţi permit să distribui o versiune modificată fără a-ţi face şi modificările “open source”. Licenţele compatibile GPL fac posibilă combinarea Pyton-ului cu alte software care au apărut sub GPL; celelate neputând face acest lucru.

Mulţumim acelor voluntari numeroşi din exteriorul echipei, care lucrând sub coordonarea lui Guido au făcut posibile aceste lansăi.

[1] Intellectual Property

[2] N.T. = termenul de lansaree găsit înanexă este traducerea termenului englezesc release

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.2

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.2 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 2.2 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001 Python Software Foundation; All Rights Reserved” are retained in Python 2.2 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 2.2 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 2.2.

4. PSF is making Python 2.2 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 2.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.2, 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 2.2, 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.