

---

# Tutorial Python

*Versiunea 2.4.1*

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

12 August 2005

**Python Software Foundation**  
Email: [docs@python.org](mailto:docs@python.org)

Copyright © 2001-2004 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See the end of this document for complete license and permissions information.

## Abstract

Python este un limbaj de programare puternic și ușor de învățat. Are structuri de date de nivel înalt puternice și o simplă, dar eficientă, abordare a programării orientate pe obiect.

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

Interpretorul Python și biblioteca standard sunt oferite sub formă de surse sau compilate pentru majoritatea platformelor existente, pe site-ul Python, <http://www.python.org/>, și pot fi distribuite gratuit. Pe același site puteți găsi distribuții Python și module scrise de alți programatori, precum și documentație adițională.

Interpretorul Python este ușor de extins prin adăugarea de noi funcții și tipuri de date implementate în C sau C++ (sau alte limbaje apelabile din C). Python poate fi folosit și drept limbaj de extensie pentru aplicații.

Acest material informează cititorul despre conceptele de bază și capabilitățile sistemului și limbajului Python. Ar fi de preferat să aveți la îndemână un interpretor Python pentru a experimenta, dar toate exemplele cuprinse în acest material pot fi înțelese fără a fi imperativă prezența interpretorului.

Pentru o descriere a obiectelor și modulelor standard, consultați documentul *Python Library Reference*. Documentul *Python Reference Manual* dă o definiție mai exactă a limbajului. Dacă doriți să extindeți sau să implementați limbajul Python într-o aplicație cu ajutorul C++ consultați documentul *Extending and Embedding the Python Interpreter* și *Python/C API Reference*. Există și cărți care acoperă limbajul Python în toată complexitatea sa.

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

După ce veți citi acest tutorial, veți putea citi și scrie module sau programe Python, și veți fi pregătiți să învățați mai multe despre diferitele biblioteci incluse în distribuția Python, descrise în documentul *Python Library Reference*.



# CUPRINS

<b>1</b>	<b>De ce Python ?</b>	<b>1</b>
<b>2</b>	<b>Utilizarea interpretorului Python</b>	<b>3</b>
2.1	Invocarea interpretorului	3
2.2	Interpretorul și mediul în care rulează	4
<b>3</b>	<b>O scurtă introducere în Python</b>	<b>7</b>
3.1	Utilizarea Python-ului drept calculator de birou	7
3.2	Primii pași în programare	16
<b>4</b>	<b>Structuri de control</b>	<b>19</b>
4.1	Instrucțiunea <code>if</code>	19
4.2	Instrucțiunea <code>for</code>	19
4.3	Funcția <code>range()</code>	20
4.4	Instrucțiunile <code>break</code> și <code>continue</code> , și clauze <code>else</code> pentru bucle	20
4.5	Instrucțiunea <code>pass</code>	21
4.6	Definirea funcțiilor	21
4.7	Mai multe despre definirea funcțiilor	23
<b>5</b>	<b>Structuri de date</b>	<b>27</b>
5.1	Mai multe despre liste	27
5.2	Instrucțiunea <code>del</code>	31
5.3	Perechi și secvențe	31
5.4	Seturi	32
5.5	Dicționare	33
5.6	Tehnici de parcurgere a secvențelor	34
5.7	Mai multe despre condiții	35
5.8	Compararea secvențelor	36
<b>6</b>	<b>Module</b>	<b>37</b>
6.1	Mai multe despre module	38
6.2	Module standard	39
6.3	Funcția <code>dir()</code>	40
6.4	Pachete	41
<b>7</b>	<b>Intrări și ieșiri</b>	<b>45</b>
7.1	Formatarea elegantă a datelor de ieșire	45
7.2	Citirea și scrierea fișierelor	47
<b>8</b>	<b>Erori și excepții</b>	<b>51</b>
8.1	Erori de sintaxă	51
8.2	Excepții	51
8.3	Tratarea excepțiilor	52
8.4	Generarea excepțiilor	54

8.5	Excepții definite de utilizator	54
8.6	Definirea acțiunilor de curățare	56
<b>9</b>	<b>Clase</b>	<b>57</b>
9.1	Câteva cuvinte despre terminologie	57
9.2	Domenii de vizibilitate (Scopes) și domenii de definiție a numelor (Name Spaces)	58
9.3	O primă privire asupra claselor	59
9.4	Alte observații	61
9.5	Moștenirea	62
9.6	Variabile private	64
9.7	Altfel de clase	64
9.8	Excepțiile pot fi clase	65
9.9	Iteratori	65
9.10	Generatori	67
9.11	Expresii generator	67
<b>10</b>	<b>Pe scurt despre Standard Library - partea I</b>	<b>69</b>
10.1	Interfața cu sistemul de operare	69
10.2	Fișierele Wildcard (asterisc)	69
10.3	Argumentele în linia de comandă	70
10.4	Redirectarea semnalării erorilor și terminarea execuției programului	70
10.5	Unificarea tratării șirurilor	70
10.6	Module matematice	71
10.7	Accesul la Internet	71
10.8	Data și timpul	71
10.9	Compresarea datelor	72
10.10	Măsurarea performanței	72
10.11	Controlul calității programului	73
10.12	Python oferit "la cheie"	73
<b>11</b>	<b>Pe scurt despre Standard Library - partea II</b>	<b>75</b>
11.1	Formatarea datelor de ieșire	75
11.2	Șirurile șablon (Templating)	76
11.3	Lucrul cu reprezentări binare ale datelor	77
11.4	Multi-threading	77
11.5	Conectivitatea (Logging)	78
11.6	Legături slabe la memorie	79
11.7	Instrumente de lucru cu listele	79
11.8	Aritmetica în virgulă flotantă zecimală	80
<b>12</b>	<b>Continuarea?</b>	<b>83</b>
<b>A</b>	<b>Editarea în linie de comandă și repetarea comenzilor anterioare</b>	<b>85</b>
A.1	Editarea în linie de comandă	85
A.2	Repetarea comenzilor anterioare (History)	85
A.3	Redefinirea tastelor funcționale	86
A.4	Comentarii	87
<b>B</b>	<b>Aritmetica în virgulă flotantă: rezultate și limitări</b>	<b>89</b>
B.1	Erori de reprezentare	91
<b>C</b>	<b>Istoricul produsului Python și licențierea</b>	<b>93</b>
C.1	Istoricul produsului	93
C.2	Terms and conditions for accessing or otherwise using Python	94
C.3	Licenses and Acknowledgements for Incorporated Software	96

## De ce Python ?

Dacă ați construit vreodată un shell complex, probabil cunoașteți senzația: vreți să adăugați încă o facilitare, dar este deja atât de lent, atât de mare, atât de complicat; sau adăugarea facilității implică un apel de funcție de sistem, sau altă funcție care nu este accesibilă decât din C. De obicei problema pe care o aveți de rezolvat nu este suficient de serioasă astfel încât să merite să rescrieți shell-ul în C; poate că rezolvarea problemei necesită folosirea șirurilor de lungime variabilă, sau a listelor sortate de nume de fișiere, lucruri accesibile într-un shell, dar care necesită multă muncă pentru a fi implementate în C; sau poate că nu sunteți suficient de familiarizat cu limbajul C.

Să considerăm o altă situație: presupunem că trebuie să lucrați cu anumite biblioteci C, și ciclul obișnuit de scriere - compilare - testare - recompilare este prea lent. Vreți să dezvoltați mult mai rapid software. E posibil să fi scris un program care poate folosi un limbaj de extensie, și nu vreți să dezvoltați un alt limbaj pentru asta, să scrieți și să verificați de erori un interpretor pentru acest limbaj, pentru ca apoi să-l adăugați la aplicația dumneavoastră.

În aceste cazuri Python este limbajul de care aveți nevoie. Este simplu de folosit, dar este un limbaj de programare adevărat, oferind posibilitatea unei mai bune structurări, și 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, și fiind un *limbaj de nivel foarte înalt*, dispune de tipuri de date de nivel înalt cum ar fi tablouri sau dicționare 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 ușor în Python ca și în aceste limbaje.

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

Există de asemenea module încorporate care oferă facilități pentru sistemul de intrări - ieșiri (I/O), apeluri de sistem, sockets, și chiar interfețe pentru sisteme de interfațare grafică utilizator (GUI) precum TK.

Python este un limbaj interpretat, care vă poate accelera munca în timpul procesului de dezvoltare, ne mai fiind nevoie să compilați codul de fiecare dată. Interpretorul poate fi folosit interactiv, ceea ce vă permite să testați foarte ușor anumite facilități ale limbajului, să scrieți imediat programe dispensabile, sau să testați anumite funcții înainte de a le folosi în programul dumneavoastră. Este foarte ușor de folosit și drept calculator de birou.

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

- tipurile de date de nivel înalt vă permit să realizați operații complexe într-o singură instrucțiune;
- gruparea instrucțiunilor este dată de paragrafare în loc de blocuri begin/end sau de acolade;
- nu este necesară declararea variabilelor.

Python este *extensibil*: dacă aveți cunoștințe de C puteți foarte ușor să adăugați o nouă funcție sau un modul, fie pentru a efectua operații la viteză maximă, fie pentru a lega programe Python la biblioteci disponibile numai în format binar (biblioteci grafice). Puteți de exemplu să adăugați interpretorul Python într-o aplicație C și să îl folosiți ca un limbaj de comandă sau de extensie pentru acea aplicație.

Numele limbajului provine de la emisiunea BBC "Monthy Python's Flying Circus" (Cercul zburător al șarpelui Monty) și nu are nici o legătură cu disgrațioasele reptile. Referințe la Monty Python în documentația ce urmează

nu numai că sunt permise, sunt chiar încurajate!

Acum că aveți o idee despre Python, poate doriți să îl examinați mai în detaliu. Din moment ce cea mai bună metodă de a învăța un limbaj de programare este utilizarea acestuia, sunteți și dumneavoastră invitați să o faceți.

În următorul capitol este explicată utilizarea interpretorului. Informațiile ce urmează pot părea banale, dar sunt esențiale pentru testarea exemplelor ce vor urma.

Cealaltă parte a tutorialului prezintă diferite facilități ale limbajului Python. Această prezentare este bogată în exemple, începând cu expresii simple, instrucțiuni, tipuri de date și terminând cu funcții, module și concepte avansate cum ar fi tratarea excepțiilor și clase definite de utilizator.



# Utilizarea interpretorului Python

## 2.1 Invocarea interpretorului

Interpretorul Python este de obicei instalat în `/usr/local/bin/python` pe mașinile unde există această cale. În lista cu directoarele de sistem a UNIX introduceți 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 și alte căi sunt posibile. Vorbiți cu administratorul dumneavoastră de sistem înainte de a lua o decizie în privința directorului de instalare. (Ex: `/usr/local/python` este o alternativă foarte populară).

Tastarea unui caracter de sfârșit de fișier (`Control-D` pe UNIX, sau `Control-Z` pe DOS sau Windows) în timpul prompt-ului principal determină oprirea execuției interpretorului cu cod de ieșire zero. Dacă acest lucru nu funcționează încercați comenzile: `import sys; sys.exit()`.

Facilitățile de editare în linie de comandă ale interpretorului nu sunt deloc deosebite. Pe sisteme UNIX, în timpul instalării Python, puteți activa opțiunea de suport a bibliotecii GNU de editare în linie de comandă. Pentru a vedea dacă această opțiune este activată, tastați combinația de taste `Control-P` la primul prompt Python. Dacă auziți un sunet, aveți activată opțiunea despre care vorbeam mai sus (consultați Anexa A, pentru o scurtă listă de combinații de taste foarte utile). Dacă nu se întâmplă nimic sau este afișat `^P`, opțiunea de editare a liniei de comandă nu este activată, deci nu veți putea decât să ștergeți (`BACKSPACE`) sau să introduceți noi caractere în linie.

Interpretorul funcționează asemănător shell-ului UNIX: când este apelat cu intrarea standard conectată la un dispozitiv TTY, citește și execută comenzi interactiv. Când este apelat cu un parametru, reprezentând un nume de fișier, citește și execută un *script* (program) din acel fișier.

O a treia modalitate de a porni interpretorul este `'python -c comandă [arg] ...'` care execută instrucțiunea/instrucțiunile din *comandă*, analog opțiunii `-c` a shell-ului. Din moment ce o comandă poate conține spații sau alte caractere utilizate de shell (ca separatori de parametri de exemplu) este bine să introduceți comanda/comenzile între ghilimele.

Observați că există o diferență între `'python file'` și `'python <file'`. În primul caz, datele de intrare ale programului sunt oferite de fișier. Din moment ce acest fișier a fost deja citit până la capăt înainte de începerea programului, programul va întâlni sfârșit de fișier imediat. În al doilea caz (cel mai interesant și cel mai folosit) datele de intrare sunt oferite de orice fișier 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 parametrul `-i` înainte de numele fișierului în care se află script-ul. Evident că acest parametru nu va funcționa dacă script-ul este citit de la intrarea standard.

### 2.1.1 Transmiterea argumentelor

Atunci când după parametrul care reprezintă fișierul în care se află script-ul, interpretorului i se mai transmit și alți parametri, acestia vor fi transmiși mai departe script-ului prin intermediul variabilei `sys.argv`, variabilă

ce conține o listă (vector) de șiruri de caractere. Atunci când în loc de numele fișierului se transmite '\_' (adică intrarea standard), `sys.argv[0] = '_'`. Când este folosită comanda `-c`, `sys.argv[0] = '-c'`. Parametrii de după `-c` comandă nu sunt considerați parametri ai interpretorului, ci sunt scriși în `sys.argv` pentru a fi folosiți 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 '>>> ') și *promptul secundar* (de obicei '. . . ') folosit pentru continuarea liniilor. Interpretorul afișează imediat după pornire un mesaj de întâmpinare în care se specifică versiunea și autorii Python-ului, urmate de primul prompt :

```
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 construcții multilinie. De exemplu:

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

## 2.2 Interpretorul și mediul în care rulează

### 2.2.1 Tratarea erorilor

Când are loc o eroare, interpretorul afișează un mesaj de eroare și starea stivei în momentul respectiv. În modul interactiv revine la prompt-ul principal. Atunci când intrările provin dintr-un fișier, interpretorul se oprește cu un cod de eroare diferit de zero după ce a afișat starea stivei. (Excepțiile tratate nu sunt erori în acest context). Unele erori sunt inevitabil fatale și cauzează terminarea anormală cu cod de ieșire diferit de zero (Ex: lipsa de memorie). Toate mesajele de eroare sunt scrise la ieșirea pentru erori (*standard error stream*). Ieșirile provenite din execuția comenzilor sunt scrise la ieșirea 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 execuției creează o excepție de tip `Keyboard Interrupt`, care poate fi tratată într-o instrucțiune `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 și apoi transformând fișierul în executabil. Caracterele ' # ! ' trebuie să fie neapărat primele două caractere din fișier.

**Observație:** Caracterul '#' este folosit pentru comentarii în Python.

Scriptul poate fi făcut executabil utilizând comanda **chmod** :

```
$ chmod +x myscript.py
```

### 2.2.3 Codificarea codului sursa

În fișierele sursă Python se pot utiliza și alte codificări în afară de ASCII. Pentru a realiza acest lucru cel mai bine este să puneți imediat după `#!` încă un caracter de comentariu de linie pentru a defini codificarea fișierului sursă :

```
# -*- coding: \code{encoding} -*-
```

Dupa această declarație toate caracterele din codul sursă vor fi tratate ca fiind codificate în *encoding* și este posibil să se scrie șiruri de caractere Unicode direct în codificarea aleasă. Lista posibilelor codificări o puteți găsi în *Python Library Reference*, secțiunea *codecs*.

De exemplu, pentru a scrie cu caractere Unicode ce includ simbolul valutar Euro, se poate utiliza codificarea ISO-8859-15, cu valoarea 164 pentru simbolul valutar Euro. Acest script va tipări valoarea 8364 (codul Unicode pentru simbolul Euro) și apoi iese :

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

currency = u" "
print ord(currency)
```

Dacă editorul folosit poate salva fișierul în format UTF-8 cu un *UTF-8 byte order mark* (aka BOM) puteți folosi acest lucru în locul unei declarații de codificare. IDLE acceptă această posibilitate dacă se setează *Option/General/Default Source Encoding/UTF-8*. **Observație:** Acest lucru nu este recunoscut de versiunile Python mai vechi de 2.2, precum și de sistemele de operare cu fișierele script ce încep cu `#!` (sistemele UNIX).

Utilizând UTF-8 (cu declarație de codificare, sau direct), caracterele utilizate se mulțumesc de limbă de glob pot apărea simultan în șiruri literale și comentarii. Nu este acceptată utilizarea caracterelor non ASCII pentru numele de identificatori. Pentru a reprezenta corect caracterele, editorul cu care lucrați ar trebui să recunoască că un fișier este scris în UTF-8, și trebuie să folosească un font care să conțină toate caracterele din fișier.

### 2.2.4 Fișierul de inițializare al modului interactiv

Când folosiți Python în modul interactiv, poate fi foarte folositor să aveți câteva comenzi care să se execute automat de fiecare dată când este pornit interpretorul. Puteți realiza acest lucru creând o variabilă de sistem cu numele `PYTHONSTARTUP` care să conțină numele unui fișier în care să aveți comenzile, care doriți să fie executate la pornirea interpretorului. Această procedură este similară, facilității *'profile'* de pe sistemele UNIX. Acest fișier de inițializare este interpretat numai când este pornit interpretorul Python în mod interactiv. Fișierul nu este interpretat nici când interpretorul Python are ca intrare un fișier, nici când sursa explicită a comenzilor este *'/dev/tty'*. Acest fișier este executat în același cadru în care se execută și comenzile interactive. În acest fișier pot fi schimbate chiar și prompt-urile `sys.ps1` sau `sys.ps2`.

Dacă doriți să executați și alte fișiere de inițializare, din directorul curent trebuie să adăugați câteva linii la fișierul principal de inițializare, utilizând un cod ca acesta `'if os.path.isfile('.pythonrc.py') : execfile('.pythonrc.py')`

Dacă doriți să folosiți acest fișier de inițializare, într-un program, trebuie să specificați acest lucru explicit în cadrul programului astfel:

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

# O scurtă introducere în Python

În exemplele ce urmează, intrările și ieșirile vor fi deosebite prin prezența sau absența prompt-urilor (principal ‘>>> ’ sau secundar ‘... ’). Pentru a testa un exemplu trebuie să scrieți linia ce apare imediat după prompt. Liniile care nu au caracterele ‘>>> ’ sau ‘... ’ în față sunt linii de ieșire, rezultate în urma comenzilor anterioare, sau a ultimei comenzi. Un prompt secundar, fără nimic după el, înseamnă că trebuie să introduceți o linie goală. Acest procedeu este folosit pentru terminarea unui bloc de instrucțiuni (de tipul ‘begin’ ... ‘end’;{ ...}).

Multe din exemplele din acest tutorial conțin comentarii. În Python comentariile încep cu un caracter ‘#’ și se termină la sfârșitul liniei. Un comentariu poate apărea la sfârșitul 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                # si al doilea comentariu
                        # ... si al treilea!
STRING = "# Acesta nu este un comentariu."
```

## 3.1 Utilizarea Python-ului drept calculator de birou

Să încercăm câteva comenzi simple Python. Porniți interpretorul și așteptați să apară prompt-ul principal ‘>>> ’.(Nu ar trebui să dureze prea mult).

### 3.1.1 Numere

Interpretorul funcționează ca un simplu calculator: puteți scrie o expresie, și ca răspuns veți primi rezultatul. Ordinea operațiilor și 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 aceiasi 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 ieșire (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 si 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+imagj*)' sau pot fi create folosind funcția '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 și cea imaginară. Pentru a afla partea reală sau cea imaginară dintr-un număr complex  $z$  folosiți `z.real` respectiv `z.imag`:

```

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

```

Funcțiile de conversie la numere reale sau întregi nu funcționează pentru numere complexe, neexistând o metodă matematică de a transforma un număr complex într-un număr real. Folosiți funcția `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 afișată este atribuită unei variabile `_`. Dacă folosiți Python ca un calculator de birou este mult mai ușor să păstrați 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 și scrisă (READ-ONLY). Nu atribuiți valori acestei variabile, deoarece acest lucru înseamnă că veți crea o variabilă nouă locală care va ascunde variabila implicită `_` și comportamentul său extraordinar.

### 3.1.2 Șiruri de caractere

În afară de numere Python poate manipula și șiruri 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.'
```

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

```
hello = "Acesta este un sir de caractere mai lung,\n\ncare se intinde pe mai mai multe linii, exact ca in 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 șir utilizând `\n`; linia nouă care urmează ultimului backslash este eliminată. Acest exemplu va afișa următoarele:

```
Acesta este un sir de caractere mai lung,
care se intinde pe mai mai multe linii, exact ca in C.
    Observati ca spatiile de la inceputul liniei au importanta.
```

Dacă construim șirul de caractere ca un șir "raw", atunci secvența `\n` nu se convertește în linie nouă, dar backslash-ul de la sfârșitul liniei și caracterul de linie nouă din sursă sunt incluse în șir ca date. Iată un exemplu :

```
hello = r"Acesta este un sir de caractere mai lung,\n\ncare se intinde pe mai mai multe linii, exact ca in C."

print hello
```

va tipari:

```
Acesta este un sir de caractere mai lung,\n\ncare se intinde pe mai mai multe linii, exact ca in C.
```

Șirurile de caractere pot exista și între perechi de cate trei ghilimele sau apostroafe ("`'''`" sau '`'''`'). În acest caz nu aveți 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 ieșire:

```

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

```

Interpretorul afișează rezultatul operațiilor cu șiruri de caractere exact în forma specificată de utilizator atunci când a introdus sau definit șirurile respectiv: între ghilimele, cu ghilimele sau alte caractere ciudate, pentru a afișa valoarea exactă a șirului. Instrucțiunea `print` despre care vă vom vorbi mai încolo poate fi folosită pentru a afișa șiruri fără ghilimele și caractere de control.

Șirurile de caractere pot fi concatenate:

```

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

```

Două șiruri 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 șiruri explicite de caractere, nu pentru orice expresie care are ca operanzi șiruri:

```

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

```

Șirurile de caractere sunt tratate ca vectori, la fel ca în C: primul caracter al unui șir are indicele 0. Nu există un tip de date separat pentru caracter. Un singur caracter este reprezentat ca un șir de lungime unu. Se pot specifica de asemenea subșiruri ale unui șir folosind doi indici, separați prin `:`. Primul indice reprezintă poziția de început a subșirului, iar cel de-al doilea indice, evident indicele în șirul principal la care se termină subșirul:

```

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

```

Dacă al doilea indice nu este specificat, se consideră că subșirul se termină la sfârșitul șirului.

Spre deosebire de C, în Python șirurile nu pot fi modificate. Atribuirea unei valori unui anumit caracter, sau unui

subșir dintr-un șir 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, crearea unui șir nou cu un conținut combinat este suficient de simplă și eficientă:

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

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

```
>>> word[:2]      # Primele doua caractere
'He'
>>> word[2:]      # Tot sirul inafara de primele doua caractere
'lpA'
```

Iată încă un exemplu:

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

Transmiterea unor parametri greșiți, 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 șirului:

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

Indicii pot fi și 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 inafara de ultimele doua caractere
'Hel'
```

Dar, atenție -0 este același lucru cu 0, și nu se numără de la dreapta!

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

Indicii negativi mai mari în valoare absolută decât dimensiunea șirului sunt trunchiați. Acest lucru nu funcționează și la selectarea unui singur caracter ( nu a unei subsecvențe ) :

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

Mai jos aveți un tabel care vă va ajuta la înțelegerea mecanismului de numerotare al unui șir 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 șirul. A doua linie reprezintă corespondența dintre caracterele șirului și indicii negativi. Un subșir de la  $i$  la  $j$  este format din toate caracterele dintre cei doi indici  $i$ , respectiv  $j$ . Pentru indicii pozitivi lungimea subșirului este dată de diferența indicilor mărginași, dacă nici unul dintre ei nu este mai mare decât lungimea șirului. De exemplu lungimea subșirului `word[1:3]` este 2.

Funcția `len()` întoarce lungimea unui șir de caractere:

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

## Vedeți și :

- *Tipurile secvență*  
(../lib/typeseq.html)  
Șirurile și șirurile Unicode descrise în secțiunea următoare sunt exemple ale tipului secvență și suportă operațiunile obișnuite pentru acest tip.
- *Metodele aplicabile șirurilor*  
(../lib/string-methods.html)  
Atât șirurilor simple, cât și Unicode li se pot aplica un număr mare de metode pentru transformare și căutare.
- *Operații de formatare a șirurilor*  
(../lib/typeseq-strings.html)  
Operațiile de formatare invocate, când șirurile obișnuite sau Unicode sunt în stânga operatorului %, sunt descrise mai în detaliu aici.

## 3.1.3 Șiruri de caractere UNICODE

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

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

La început au existat numai 256 de numere fiecare având asociat un caracter, și toate textele erau asociate unei "pagini de cod"(code page) care făcea legătura între caractere și numerele asociate. Acest lucru a creat foarte multă confuzie odată cu internaționalizarea software-ului (scriere usuală : 'i18n' — 'i' + 18 characters + 'n'). Unicode rezolvă aceste probleme asociând o singură pagină tuturor script-urilor.

Crearea șirurilor de caractere Unicode este la fel de simplă ca și crearea șirurilor obișnuite :

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

Caracterul 'u' din fața ghilimelelor șirului indică interpretorului că trebuie să creeze un șir Unicode din expresia care urmează. Dacă doriți să includeți în șir caractere speciale, o puteți face foarte simplu folosind codarea Python *Unicode-Escape*. Iată un exemplu:

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

Secvența escape `\u0020` indică interpretorului să insereze caracterul cu ordinalul 0x0020 (caracterul spațiu) în poziția precizată. Alte caractere sunt utilizate folosind direct valoarea ordinalului său ca ordinal Unicode. Datorită faptului că primele 256 de caractere Unicode sunt aceleași ca și în notația standard Latin-1, folosită în multe țări occidentale, procesul de introducere a caracterelor Unicode devine mult mai simplu.

Pentru experți există și un mod brut (raw-mode) identic cu cel utilizat pentru șirurile normale. Înaintea ghilimelelor șirului 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 !'
```

Mod raw poate fi foarte util dacă aveți de introdus un număr mare de backslash-uri.

În afară de metodele de codare standard, Python oferă multe alte metode de a crea șiruri Unicode, pe baza codărilor cunoscute.

Funcția predefinită `unicode()` permite accesarea tuturor CODEC-urilor Unicode (CODEC = COder & DE-Coder). Câteva dintre codificările foarte cunoscute ale caror CODEC-uri au fost convertite sunt : *Latin-1*, *ASCII*, *UTF-8* și *UTF-16*. Ultimele două permit stocarea caracterelor Unicode pe unul sau mai mulți octeți. Codificarea implicită este ASCII, care permite trecerea caracterelor cuprinse între 0 și 127, dar blochează celelalte caractere semnalizând eroare. Când un șir Unicode este tipărit, scris într-un fișier, sau convertit cu funcția `str()`, conversia pornește utilizând 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 șir Unicode într-un șir pe 8 biți 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"ăöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Dacă aveți datele într-o anumită codificare și doriți să obțineți un șir Unicode corespunzător din ele, folosiți funcția `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 dispoziția 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, și aflată între paranteze pătrate. Elementele unei liste nu trebuie să fie neapărat de același tip:

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

La fel ca și la șiruri, primul element al unei liste are indicele 0. În același fel ca și la șiruri, listele pot fi poziționate, concatenate și așa 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 de șiruri, elementele unei liste pot fi modificate:

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

Funcția `len()` se aplică și listelor :

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

Utilizarea sublistelor este de asemenea posibilă. Prin folosirea lor se poate modifica chiar și 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 creați 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')      # Vezi sectiunea 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']

```

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

## 3.2 Primii pași în programare

Bineînțeles că putem folosi Python și la alte lucruri decât pentru a aduna 2 cu 2. Putem de exemplu să generăm o subsecvență din șirul 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 noțiuni:

- Prima linie conține o *atribuire multiplă*. Variabilele *a* și *b* iau valorile 0 respectiv 1. Pe ultima linie, de asemenea este folosită atribuirea multiplă. Evaluarea expresiilor din partea dreaptă a unei atribuirii 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 condiția este adevărată). La fel ca în C, zero înseamnă fals, și orice număr diferit de zero înseamnă adevărat. Condiția poate fi un șir, un element de listă, absolut orice. Orice secvență de lungime diferită de zero înseamnă adevărat, și invers. Condiția folosită în acest exemplu este o comparație. Comparatorii standard sunt la fel ca în C: *<*(mai mic), *>*(mai mare), *==*(egal), *<=*(mai mic sau egal), *>=*(mai mare sau egal), *!=*(diferit).
- Instrucțiunile din buclă sunt *aliniate*. Alinierea (indentation) reprezintă modul în care Python grupează instrucțiunile. Deocamdată (!) Python nu dispune de un editor în linie de comandă inteligent, astfel încât alinierea, tabularea să se facă automat. Sunteți deci obligat să folosiți tab sau spații pentru a realiza gruparea instrucțiunilor. Când veți scrie programe Python veți folosi un editor de text. Majoritatea editoarelor de la ora actuală realizează automat tabularea. Pentru a încheia un bloc de instrucțiuni trebuie introdusă o linie goală pentru a indica editorului că editarea blocului de comenzi s-a încheiat (editorul nu poate ghici când ați introdus ultima linie. Atenție ! fiecare instrucțiune dintr-un bloc de instrucțiuni trebuie să aibă aceleași număr de spații și taburi înainte, deci instrucțiunile trebuie să fie **perfect** aliniate.
- Instrucțiunea *print* afișează expresia pe care o primește ca parametru. Șirurile de caractere sunt afișate fără ghilimele și cu un spațiu este inserat între elemente, astfel încât afișarea diverselor valori să meargă de la sine:

```

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

```

O virgulă la sfârșitul listei de parametri ai instrucțiunii *print*, inhibă saltul cursorului pe linia următoare la sfârșitul instrucțiunii:

```

>>> 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

```

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





## Structuri de control

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

### 4.1 Instrucțiunea `if`

Poate că cea mai cunoscută instrucțiune de control este instrucțiunea `if`. Exemplu:

```
>>> x = int(raw_input("Introduceți un numar intreg : "))
>>> if x < 0:
...     x = 0
...     print 'Negativul schimbat in zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Unul singur'
... else:
...     print 'Mai multi'
... 
```

Pot exista mai multe secțiuni `elif` sau niciuna, iar secțiunea `else` este opțională. Cuvântul cheie `'elif'` este evident prescurtarea de la `'elseif'`, și este folositor pentru a evita tabularea excesivă. O secvență `if..elif..elif` funcționează ca un bloc `case` sau `switch`, secvențe proprii altor limbaje.

### 4.2 Instrucțiunea `for`

Instrucțiunea `for` din Python diferă un pic față de ce a-ți întâlnit în C sau Pascal. În loc de o iterație dată de o progresie aritmetică (Pascal), sau de o iterație foarte flexibilă, pentru care programatorul poate defini atât pasul iterației, cât și condiția de oprire (C), iterațiile instrucțiunii Python `for` funcționează după elementele unei secvențe (sir sau listă).

```
>>> # Masoara marimea unor siruri
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Nu este normal (și sigur) ca secvența iterată să fie modificată în timpul secvenței `for` (este cazul numai al secvențelor modificabile, cum ar fi listele). Dacă apare necesitatea de a modifica secvența în timpul iterației,

iterația trebuie să fie asociată unei copii a secvenței. Notăția subsecvențelor realizează o particularitate convenabilă :

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

### 4.3 Funcția range( )

Dacă este necesară o iterație pe o mulțime de numere, puteți folosi funcția range( ) pentru a genera liste ce conțin progresii aritmetice :

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

Parametrul furnizat funcției range( ) nu va fi niciodată un membru al secvenței. Este posibil ca funcția range( ) să genereze o secvență începând cu un alt număr decât 0, sau rația 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 iterație pe o mulțime de numere folosiți funcțiile range( ) și 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 Instrucțiunile break și continue, și clauze else pentru bucle

La fel ca în C, instrucțiunea break termină forțat orice buclă while sau for. Instrucțiunea continue trece necondiționat la următoarea iterație. Instrucțiunile iterative pot avea și o clauză else. Instrucțiunile din cadrul unei astfel de clauze else sunt executate atunci când bucla se termină odată cu terminarea listei (for) sau atunci când condiția buclei devine falsă (pentru while) aceste instrucțiuni nu sunt executate dacă bucla este terminată printr-o instrucțiune 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 fara sa gaseasca 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 Instrucțiunea pass

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

```

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

```

## 4.6 Definirea funcțiilor

Putem crea o funcție care generează șirul lui Fibonacci până la o limită arbitrară:

```

>>> def fib(n): # Scrie sirul lui Fibonacci pana la n
...     "Scrie sirul lui Fibonacci pana la n"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Folosim functia 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 *definiția* unei funcții. Acestea trebuie să îi urmeze numele funcției și, în paranteză, o listă de parametri. Instrucțiunile care formează funcția încep pe linia imediat următoare și trebuie să fie distanțate mai mult decât antetul funcției față de margine. Prima instrucțiune din corpul funcției poate fi, opțional, un șir de caractere. Acest șir reprezentând documentația funcției (*docstring* = șirul de documentație al funcției). Există utilitare care generează documentație pornind de la aceste șiruri, deci este o practică bună să vă folosiți de această facilitare.

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

Cu toate acestea variabilelor globale nu li se pot atribui valori în cadrul unei funcții, decât dacă se folosește

instrucțiunea `global`.

Parametrii actuali ai unei funcții apelate sunt introduși în tabela de simboluri locală a acelei funcții în momentul apelării ei. Astfel transferul argumentelor se face utilizând *apelul prin valoare* (unde *valoare* este totdeauna un obiect *referință*, nu valoarea obiectului).<sup>1</sup>

Când o funcție apelează o alta funcție, este creată o nouă tabelă de simboluri pentru funcția apelată.

Definirea unei funcții trece numele funcției în tabela de simboluri curentă.

Funcția nou creată este recunoscută de interpretor ca o funcție definită de utilizator. În Python există un mecanism de redenumire a funcțiilor:

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

Ați putea obiecta că funcția `fib` este o procedură nu o funcție. În Python, la fel ca în C, procedurile sunt funcții, numai că nu returnează nici o valoare. Tehnic vorbind, procedurile returnează totuși o valoare, aceasta este `None` (niciuna). `None` este un cuvânt cheie (predefinit). În mod normal valoarea `None` nu este afișată de interpretor. Totuși dacă doriți să vă convingeți că așa funcționează procedurile încercați următorul exemplu:

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

Este foarte simplu să construim o funcție care întoarce ca rezultat șirul lui Fibonacci:

```
>>> def fib2(n): # Intoarce sirul lui Fibonacci pana la n
...     "Intoarce o lista continand sirul 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 functiei
>>> f100             # afisarea rezultatului
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Acest exemplu demonstrează mai multe facilități ale limbajului Python:

- Instrucțiunea `return` întoarce rezultatul funcției. `return` fără nici un parametru întoarce ca rezultat al funcției valoarea `None`. Puteți folosi `return` fără parametrii pentru a încheia execuția unei funcții înainte de a se ajunge la finalul instrucțiunilor din cadrul funcției. Valoarea `None` este de asemenea întoarsă ca rezultat al unei funcții atunci când au fost executate toate instrucțiunile procedurii.
- Instrucțiunea `result.append(b)` apelează o *metodă* a obiectului `result`. O metodă este o funcție care ‘aparține’ unui obiect. Metodele sunt apelate folosind sintaxa `obj.metodă` (unde `obj` este un obiect, poate fi și o expresie, iar `metodă` este numele metodei ce este apelată). În funcție 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 creați obiecte și metode folosind clasele, dar despre clase vom discuta mai târziu. Metoda `append` este definită pentru obiecte de tip listă, și adaugă un element la sfârșitul listei. În acest exemplu ar fi echivalentă cu `result = result + [b]` dar mult mai eficientă.

---

<sup>1</sup> De fapt *apelul prin referință* poate fi mai bine înțeles dacă este transmis un obiect modificabil și apelantul va vedea dacă în urma apelării obiectul s-a modificat ( ex.: un element inserat într-o listă).

## 4.7 Mai multe despre definirea funcțiilor

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

### 4.7.1 Valori implicite pentru argumente

Cea mai utilă metodă este să specificați valori implicite pentru argumente. În acest fel creați o funcție care poate fi apelată cu mai puține 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
```

Funcția definită mai sus poate fi apelată în două moduri:

`ask_ok('Sunteți sigur că doriți să ieșiți ?')` sau: `ask_ok('Doriți să ștergeți fișierul ?', 2)`

Valorile implicite sunt evaluate în timpul interpretării definiției funcției, deci următoarea secvență

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

va afișa 5.

**ATENȚIE!!!:** Valorile implicite sunt evaluate o singură dată. Aveți grijă atunci când valoarea implicită este un obiect modificabil, cum este lista, dicționarul, sau instanțierea unei clase. De exemplu funcția 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 afișa

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

Dacă nu doriți ca valoarea implicită să fie reținută după terminarea funcției, puteți 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

Funcțiile pot fi apelate folosind în loc de argumente cuvinte cheie de forma: `cuvant_cheie = valoare`. Pentru exemplificare observați funcția:

```
def parrot(voltage, state='mort', action='zboara', type='Norwegian 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 voltage
parrot(voltage=5.0, 'mort') # Dupa cuvantul cheie trebuie sa urmeze
un argument tip cheie
parrot(110, voltage=220) # Doua valori atribuite aceleiasi varibile
parrot(actor='John Cleese') # Cuvant cheie necunoscut
```

În general o listă de argumente trebuie să aibe oricâte argumente poziționale, 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 poziționale nu pot fi folosite drept cuvinte cheie în cadrul aceluiași 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`, funcția va primi o listă care va conține parametrii de tip cuvânt cheie. Dacă se folosește un parametru de tipul `*nume`, funcția va primi o listă conținând argumentele suplimentare în afara celor formale. Dacă sunt folosiți împreună parametrul `*nume` trebuie să se afle înaintea celui de tip `**nume`. Totul poate părea ambiguu dar veți fi lămuiriți 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 funcții 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 funcție poate fi apelată cu un număr arbitrar de argumente. Înainte de argumentele arbitrare (opționale) pot exista mai multe argumente normale:

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

### 4.7.4 Despachetarea listelor de argumente

Situația inversă este aceea în care argumentele se găsesc deja într-o listă sau pereche, dar este necesară despachetarea lor pentru apelarea unei funcții care necesită argumente separate pozițional. De exemplu, funcția internă `range()` necesită parametri separați pentru `start` și `stop`. Dacă argumentele nu sunt transmise separat, scrieți apelarea funcției folosind operatorul `*` pentru despachetarea argumentelor dintr-o listă sau dintr-o pereche :

```
>>> range(3, 6)                # apelare normala
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)               # apelare cu despachetarea argumentelor dintr-o lista
[3, 4, 5]
```

### 4.7.5 Forme Lambda

La cererea utilizatorilor de Python, au fost adăugate câteva facilități specifice limbajelor funcționale și Lisp-ului. Folosind cuvântul cheie `lambda` puteți crea mici funcții. Iată o funcție care întoarce ca rezultat suma argumentelor: `'lambda a, b : a+b'`. Formele lambda pot fi folosite acolo unde sunt necesare funcții obiect.

Aceste funcții 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
>>>
```

#### 4.7.6 Șirurile de documentație

Există anumite convenții privind conținutul și formatarea șirurilor de documentație.

Prima linie ar trebui să fie scurtă și foarte concisă. Pentru concizie nu ar trebui precizat numele obiectivului sau tipul, acestea fiind subînțelese. Această linie trebuie să înceapă cu literă mare și 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 amănunțită. Următoarele linii ar trebui să descrie mai pe larg obiectul, semnificația parametrilor, etc.

Interpretorul Python ia în considerație spațiile din șirurile de caractere împărțite pe mai multe linii, din această cauză pentru a avea o documentație aranjată "frumos" trebuie să vă folosiți 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!
```



# Structuri de date

Acest capitol subliniază anumite lucruri deja cunoscute și 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)** Aduagă un element la sfârșitul listei.

**extend(L)** Aduagă la sfârșitul listei, elementele listei furnizate ca parametru.

**insert(i, x)** Inserează un element într-o anumită poziție. Primul argument reprezintă indicele elementului din listă înaintea căruia se va face inserția, 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)** Șterge din listă primul element găsit cu valoarea 'x'. Dacă nu există un astfel de element apare o eroare.

**pop([i])** Șterge din listă elementul de pe poziția 'i', și întoarce valoarea acestuia. Dacă nu este specificat nici un parametru `a.pop()`, va șterge și va returna ultimul element din listă.

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

**count(x)** Întoarce numărul de apariții ale valorii 'x' între elementele listei.

**sort()** Sortează elementele listei.

**reverse()** Schimbă ordinea elementelor din listă.

Iată un exemplu care folosește 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 ușor pentru a simula comportamentul unei stive. Pentru a adăuga un element pe stivă (PUSH), puteți folosi `append()`. Pentru a scoate un element din stivă (POP), folosiți `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

Puteți 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 folosiți `append()`, iar pentru a extrage un element folosiți `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 funcțională

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

Funcția `filter()`, cu sintaxa `filter(funcție, secvență)`, întoarce o secvență formată din elementele secvenței specificate ca parametru, care îndeplinesc condiția testată de *funcție*. 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]

```

Funcția `map()`, cu sintaxa `map(funcție, secvență)`, apelează funcția specificată ca parametru pentru fiecare element din secvență, și întoarce o nouă listă formată din rezultatele întoarse de funcție. Pentru a calcula pătratele unor numere dintr-o listă puteți folosi o secvență 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]

```

Funcția `map()` poate accepta ca parametrii mai multe secvențe. În acest caz funcția transmisă ca parametru trebuie să fie modificată corespunzător, astfel încât să accepte același număr de parametrii câte secvențe sunt transmise. Dacă secvențele sunt diferite ca lungime, atunci când una dintre secvențe s-a terminat, în loc de un element funcției i se transmite `None`. Dacă în loc de un nume de funcție, se transmite funcției `map()` `None`, atunci funcția va fi înlocuită cu o funcție care va întoarce ca rezultat parametrul primit. Puteți folosi comportamentul funcției `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)]

```

Funcția `reduce(funcție, secvență)` întoarce o valoare simplă care este calculată în felul următor: este apelată funcția (obligatoriu o funcție binară care acceptă numai 2 parametrii), cu parametrii primul și al doilea termen al secvenței. Funcția întoarce un rezultat care împreună cu al treilea element sunt transmise din nou funcției, care generează un alt rezultat și așa 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 secvență va fi returnată valoarea acestuia. Dacă lista este goală, va fi generată o excepție. Funcția `reduce()` poate primi și un al treilea parametru care semnifică valoarea de la care începe calculul. În acest caz primul apel al funcției are ca parametrii această valoare și primul element al listei. Dacă funcția `reduce()` este apelată cu trei parametri, iar lista este goală, aceasta va întoarce ca rezultat al treilea parametru. Iată un exemplu care ilustrează modul de lucru al funcției `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 funcțiilor `map()`, `filter()` sau `lambda()`. Definiția listei este, de cele mai multe ori, mult mai clară decât cea obținută prin alte metode. Acest mod generalizat, de a genera, a defini liste constă în asocierea unei expresii, cu o clauză `for` și cu niciuna sau mai multe clauze `for` sau `if`. Rezultatul va fi o listă care provine din evaluarea expresiei în contextul clauzelor `for` și `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]
>>> [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 Instrucțiunea del

Există o metodă de a șterge un element dintr-o listă specificând indicele elementului în loc de valoarea elementului. Această metodă poate fi folosită și pentru a șterge porțiuni dintr-o listă. Toate acestea pot fi realizate utilizând instrucțiunea `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]

```

Instrucțiunea `del` poate fi folosită și pentru a șterge variabile, de exemplu o întreagă listă:

```

>>> del a

```

Vom vedea mai târziu și alte modalități de a utiliza instrucțiunea `del`.

## 5.3 Perechi și secvențe

<sup>1</sup> Am observat până acum că listele și șirurile de caractere au multe caracteristici comune, de exemplu: indexarea și poziționarea (slicing). Șirurile și listele sunt două exemple de tipuri de date secvență. Deoarece Python este un limbaj care evoluează, în timp pot fi adăugate și alte tipuri de date secvență. Există și un alt tip standard de date: perechea (enumerarea).

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

<sup>1</sup> N.T. termenul românesc **PERECHE** este impropriu, forma originală fiind **TUPLE**.

```

>>> 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 puteți vedea perechile(enumerările) sunt afișate 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 angajaților dintr-o bază de date, etc. Puteți face o analogie (atâta timp cât rămâne numai o analogie menită să vă ajute la înțelegerea acestui tip de date) între tipurile de date compuse ce pot fi definite cu `struct` din C++, și perechile din Python. Perechile, la fel ca șirurile nu pot fi modificate: nu pot fi atribuite valori unui anumit element al unei perechi (puteți însă să simulați o modificare folosind alte metode). Este posibil să creați perechi care conțin 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 conținută în pereche urmată de o virgulă. Cam neplăcut, dar funcționează:

```

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

```

Instrucțiunea `t = 12345, 54321, 'Salut !'` este un exemplu de împachetare. Valorile 12345, 54321 și 'Salut !' sunt ”împachetate” împreună. Operația inversă este de asemenea posibilă:

```

>>> x, y, z = t

```

Această operație este numită suficient de elocvent ”despachetarea unei secvențe”. Despachetarea unei secvențe necesită ca lista de variabile de la stânga atribuirii să conțină tot atâtea variabile cât secvența împachetată de la dreapta. De observat este faptul că îmbricarea nu este decât o combinație succesivă de împachetări sau despachetări! Ingenios, nu?!

Există o asimetrie aici: în timp ce o împachetare va genera întotdeauna o pereche, despachetarea funcționează pentru orice fel de secvență.

## 5.4 Seturi

Limbajul Python are și un tip de date pentru *seturi*. Un set este o colecție neordonată de elemente unice (fără duplicate). Esența ei constă în testarea membrilor și eliminarea duplicatelor. Obiectele de tip set suportă, de asemenea, operații matematice ca : uniunea, intersecția, diferența și diferența simetrică.

Iată o scurtă demonstrație :

```

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruits = set(basket)                      # creaza un set fara duplicate
>>> fruits
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits                        # testarea rapida a membrilor
True
>>> 'crabgrass' in fruits
False
>>> # Dmonstrarea operatiilor set pentru literele unice a doua cuvinte
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                         # literele unice din a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                                     # literele unice din b
set(['r', 'd', 'b'])
>>> a | b                                     # literele din a sau b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                                     # literele atat din a cat si din b
set(['a', 'c'])
>>> a ^ b                                     # literele din a sau b , dar nu din amandoua
set(['r', 'd', 'b', 'm', 'z', 'l'])

```

## 5.5 Dicționare

Un alt tip de date predefinit în Python și care se poate dovedi foarte folositor este *dicționarul*. Dicționarele pot fi întâlnite și în alte limbaje, sub alte nume, cum ar fi "memorii asociative" sau "valori asociative". Spre deosebire de secvențe (liste, șiruri, etc.) care sunt indexate cu numere (indicii sunt numere), dicționarele sunt indexate prin *chei*, care pot fi definite de oricare din tipurile de date invariabile(nemodificabile), de exemplu: șiruri de caractere sau numere. Perechile pot fi folosite drept chei ale unui dicționar numai dacă conțin obiecte invariabile. Nu puteți folosi drept chei listele deoarece acestea pot fi modificate folosind metode ca `append()` sau `extend()`.

Este indicat să vă gândiți la un dicționar ca la o mulțime neordonată de perechi cheie-valoare, cu observația că o cheie trebuie să fie unică într-un dicționar. O pereche de acolade crează un dicționar gol: `{}`. Puteți crea un dicționar dacă între acolade introduceți o listă de perechi *cheie:valoare*, ale cărei elemente sunt separate prin virgulă, : `dicționar = {'jack':4098, 'Sape':4139}`. Acea este, de asemenea, o modalitate de a scrie dicționare la dispozitivul de ieșire. Operațiile principale pe care le realizează un dicționar sunt: stocarea unei valori cu anumită cheie și extragerea unei valori cunoscându-se o anumită cheie. O pereche cheie-valoare poate fi ștearsă folosind instrucțiunea `del`. Dacă se adaugă o valoare în dicționar pentru o cheie care există deja, valoarea veche asociată acelei chei este pierdută. O eroare este generată, bineînțeles, dacă veți încerca să extrageți o valoare pentru o cheie inexistentă.

Metoda `keys()` a unui obiect dicționar întoarce o listă cu toate cheile existente în respectivul dicționar. Lista returnată nu este sortată, însă pentru a o sorta puteți folosi metoda `sort()` a obiectului listă returnat de funcție. Pentru a verifica dacă o anumită cheie se află deja în dicționar folosiți 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

```

## 5.6 Tehnici de parcurgere a secvențelor

Când se parcurge un dicționar, cheia și valoarea corespunzătoare se pot evidenția simultan utilizând metoda `iteritems()`.

```

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave

```

Când se parcurge o secvență, indexul poziției și valoarea corespunzătoare se pot găsi simultan utilizând funcția `enumerate()`.

```

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

```

Pentru a parcurge două sau mai multe secvențe în același timp intrările se pot împerechea cu funcția `zip()`.

```

>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your %s?      It is %s.' % (q, a)
...
What is your name?      It is lancelot.
What is your quest?     It is the holy grail.
What is your favorite color?  It is blue.

```

Pentru a parcurge o secvență în ordine inversă, întâi se specifică secvența în ordine normală, apoi se apelează funcția `reversed()`.



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

Pentru a bucla o secvență în ordine sortată, se folosește funcția `sorted()`, care returnează o nouă listă sortată, dar sursa rămâne nealterată.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

## 5.7 Mai multe despre condiții

Condițiile folosite în cadrul instrucțiunilor `while` sau `if` pot conține și alți operatori decât cei de comparație.

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

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

Condițiile pot avea structuri foarte complexe dacă sunt folosiți operatorii `and`, `or` sau `not`. Operatorii de comparație au aceeași prioritate care este mai mică decât aceea a operatorilor numerici. Operatorii logici, booleani, `and`, `or`, `not` au cea mai mică prioritate, mai mică decât aceea a operatorilor de comparație. 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`”. Bineînțeles, pentru claritate, puteți folosi paranteze pentru a combina condițiile și.

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

Este posibilă atribuirea unei valori rezultate dintr-o comparație, sau orice altă condiție, 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 mulțime de erori întâlnite în programele C, de exemplu `=` în loc de `==`.

## 5.8 Compararea secvențelor

Obiectelor de tip secvență pot fi comparate cu obiecte secvență de același tip. Comparările între secvențe folosesc principiile ordonării *lexicografice*: se compară întâi primul element din fiecare listă. Dacă diferă este afișat rezultatul, dacă sunt egale se trece la compararea următoarelor două elemente, și așa mai departe până la epuizarea uneia dintre liste. Dacă elementele ce urmează a fi comparate sunt la rândul lor liste, compararea lexicografică are loc recursiv. Dacă toate elementele celor două liste sunt egale se consideră că listele sunt egale. Dacă una dintre secvențe este o subsecvență inițială a celeilalte, atunci secvența mai scurtă este cea mai mică. Ordonarea lexicografică pentru șiruri folosește ordinea ASCII a caracterelor. Exemple de comparații între secvențe de același 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)
```

Python permite și 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 șir, un șir mai mic decât o pereche, etc. Comparațiile între numere se fac luând în considerație valoarea numerică chiar dacă numerele sunt de tipuri diferite, și ca atare 0 va fi egal cu 0.0, etc. <sup>2</sup>

---

<sup>2</sup>Regulile de comparare între obiecte de tipuri diferite nu ar trebui considerate de foarte mare încredere; acestea pot fi schimbate la versiuni viitoare ale limbajului.

# Module

Dacă ieşiţi din interpretorul Python şi intraţi din nou, definiţiile pe care le-aţi făcut (funcţii, variabile, 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 apoi 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 crea fişierul `"fibonacci.py"`:

```
# modul cu numerele lui Fibonacci

def fib(n):      # scrie sirul lui Fibonacci pana la n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):     # intoarce sirul 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 conține atât instrucțiuni executabile cât și definiții de funcții. Instrucțiunile sunt menite să realizeze inițializarea modului și se execută *o singură dată*, atunci când modulul este importat<sup>1</sup>.

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

Cu toate astea, autorul unui modul se poate folosi de variabilele globale, fără a-și face griji pentru eventuale coliziuni cu variabilele globale ale mediului în care modulul va fi importat. Pe de altă parte dacă aveți neapărat nevoie puteți modifica direct variabilele globale ale unui modul, folosind aceeași convenție ca la apelarea funcțiilor unui modul: `modul.variabilă`.

Modulele pot importa alte module. Este indicat ca instrucțiunile `import` pot fi plasate la începutul programului și, dar nu este obligatoriu.

Există o metodă de a încărca definițiile unui anumit modul direct în tabela de simboluri globală a modului 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 modului nu a fost introdus în tabela de simboluri locală (`fibo` nu este definit). Pentru a încărca toate definițiile modului direct în tabela locală folosiți sintaxa :

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

Acum în tabela locală au fost încărcate toate numele de variabile și funcții din modul, mai puțin cele care încep cu caracterul `__`.

### 6.1.1 Calea în care sunt căutate modulele

Atunci când un modul numit `'xxx'` este importat, interpretorul caută un fișier numit `'xxx.py'` în directorul curent, și în toate directoarele specificate în variabila de stare `PYTHONPATH`. Aceasta are aceeași sintaxa ca și variabila de sistem `PATH`, care este o listă de directoare. Dacă variabila `PYTHONPATH` nu este definită, sau modulul nu este găsit în directoarele din lista, atunci va fi căutat în directorul implicit de instalare, de obicei, în UNIX, `'/usr/local/lib/python'`.

### 6.1.2 Fișiere Python "compile"

Un important aport de viteză la pornirea programelor care încarcă o mulțime de module standard este adus de următorul comportament al Python-ului: dacă în directorul unde este găsit fișierul `'xxx.py'` se mai află și `'xxx.pyc'`, se presupune că acest ultim fișier conține o variantă compilată a modului și este încărcat în loc de `'xxx.py'`.

<sup>1</sup>De fapt declarațiile de funcții sunt tot 'instrucțiuni' care sunt 'executate' : execuția înseamnă încărcarea numelui funcției în tabela de simboluri globale a modului.

Timpul la care a fost modificat ultima oară fișierul `.py` este înregistrat în fișierul `.pyc` în momentul în care este creat, iar în momentul importului dacă aceste momente de timp nu corespund fișierului `.pyc` este ignorat.

De obicei nu trebuie să faceți nimic deosebit pentru a crea fișierul `xxx.pyc`. Ori de câte ori `xxx.py` este compilat cu succes este creat sau suprascris și fișierul `xxx.pyc`. Dacă scrierea fișierului compilat nu reușește veți fi avertizat printr-un mesaj de eroare. Conținutul fișierelor de tip `.pyc` este independent de platformă, astfel încât Python devine un limbaj foarte portabil.

Câteva indicații pentru experți:

- Când interpretorul Python este apelat folosind opțiunea **-O**, se generează cod optimizat care va fi stocat în fișiere de tip `.pyo`. La momentul actual optimizatorul nu este de prea mare folos, nu face decât să elimine din cod instrucțiunile `assert` și `SET-LINENO`. Atunci când se folosește această opțiune tot codul este optimizat, fișierele `.pyc` sunt ignorate iar fișierele `.py` sunt compilate și optimizate.
- Dacă interpretorul Python este lansat în execuție folosind opțiunea **-OO**, compilatorul va genera cod optimizat, care în cazuri rare să ducă la funcționări eronate ale programului. În mod curentă se elimină numai șirurile `__doc__` din codul compilat, rezultând fișiere `.pyo` mai compacte. Din moment ce unele programe pot folosi aceste șiruri, eliminarea lor ar putea genera erori. În concluzie, ar trebui să folosiți această opțiune numai atunci când știți foarte bine ce faceți.
- Un program nu funcționează mai rapid, atunci când este citit dintr-un fișier de tip `.pyc` sau `.pyo`, față de unul `.py`. Singurul lucru care diferă este viteza de încărcare a programelor în memorie.
- Atunci când este rulat un script, prin lansarea interpretorului având ca parametru numele fișierului în care se află script-ul, codul script-ului nu este compilat și nu se generează fișiere de tip `.pyc` sau `.pyo`. În cazul în care aveți un script de dimensiuni mari, și doriți să creșteți viteza de încărcare, puteți să mutați bucăți din cod într-un modul pe care să îl importați apoi în script. Interpretorul Python poate fi lansat și folosind direct un fișier cu extensia `.pyc` sau `.pyo`.
- Este posibil să lansați interpretorul Python doar cu un fișier de tip `xxx.pyc` (sau `xxx.pyo`) dacă se folosește **-O**, fără ca fișierul `xxx.py` asociat să existe. Puteți folosi această facilități atunci când distribuiți o bibliotecă într-un format de cod greu de decriptat.
- Modulul `compileall` poate fi folosit pentru a crea fișiere `.pyc` (sau `.pyo`, utilizând **-O**) pentru toate modulele dintr-un director.

## 6.2 Module standard

Python dispune de o bibliotecă de module standard, a căror descriere o puteți găsi în "Python Library Reference". Unele module sunt integrate în interpretor, deși nu fac parte din nucleul limbajului, din definiția acestuia, dar sunt integrate pentru eficiență sau pentru a facilita accesul la primitivele sistemului de operare. Setul de module integrate este o opțiune de configurare, care de asemenea depinde de particularitatea platformei. De exemplu modulul `amoeba` este disponibil pe sisteme care pun la dispoziție primitive Amoeba. Un anumit modul necesită o atenție specială: `sys` care este integrat în orice interpretor Python. Variabilele `sys.ps1` și `sys.ps2` definesc șirurile de caractere ce vor fi folosite de interpretor pentru prompt-ul principal și 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` conține o listă de directoare unde Python caută module. Această variabilă este inițializată din variabila de sistem `PYTHONPATH`, sau cu o valoare implicită în cazul în care `PYTHONPATH` nu este definită. Puteți modifica `sys.path` folosind operații specifice listelor:

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

## 6.3 Funcția `dir()`

Funcția integrată `dir()` poate fi folosită pentru a determina ce nume (de variabile, funcții, etc.) definește un modul. Rezultatul acestei funcții este o listă sortată de șiruri de caractere:

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

Dacă funcția `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']
```

Observați că listează diferite tipuri de nume: variabile, funcții, module, etc.

`dir()` nu listează și numele de funcții sau variabile integrate. Dacă doriți o astfel de listă, puteți apela `dir(__builtin__)`, unde `__builtin__` este modulul în care se află funcțiile și variabilele integrate:

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

## 6.4 Pachete

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

Să presupunem că doriți să proiectați o colecție de module (un pachet) pentru manipularea fișierelor de sunet, și a sunetelor. Există o multitudine de formate de sunet (de obicei recunoscute după extensia lor '.wav', '.aiff', '.au') așa că este necesară crearea și întreținerea unei colecții de module în continuă creștere care să permită conversia între diverse formate de fișiere. Există multe alte operații pe care poate ați dori să le executați (mixaje, adăugarea ecoului, efecte stereo, etc.), așa că, de fapt, veți scrie o serie nesfârșită de module care să permită aceste operații. Aveți mai jos o posibilă structură a pachetului dumneavoastră (în sensul unei ierarhii de fișiere):

Sound/		Pachet "tata"
__init__.py		Initializarea pachetului 'sound'
Formats/		Subpachet pentru conversii intre fisiere
__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 submodulul `Sound.Effects.echo`. O funcţie conţinută de submodul trebuie apelaată 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
```

Aici se încarcă, de asemenea, submodulul `echo`, dar funcţia `echofilter()` poate fi accesat direct :

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



De observat că atunci când se folosește sintaxa `from packet import element`, elementul poate fi, fie un sub-modul (sau subpachet), fie un nume definit în pachet, de exemplu un nume de funcție sau variabilă, sau un nume de clasă.

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

Când folosiți o sintaxă de genul: `import element.subelement.subsubelement`, fiecare element, mai puțin ultimul trebuie obligatoriu să fie un pachet. Ultimul element poate fi un modul, un pachet, dar nu poate fi o clasă, o funcție, 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 instrucțiune de genul: `from Sound.Effects import *`? Într-un caz ideal, programatorul ar putea să spera că interpretorul Python va realiza o căutare recursivă în sistemul de fișiere și directoare după modulele existente în pachet. Din nefericire nu funcționează foarte bine pe platforme Mac sau Windows, din varii motive. Pe astfel de platforme existența unui fișier `'ECHO.PY'` nu garantează că în acest fișier se află un modul numit `Echo`, cât și `echo` sau `echo`, și așa mai departe. Spre exemplu Windows 95 are prostul obicei de a afișa numele de fișiere cu prima literă mare. Sistemul de fișiere DOS 8+3 ridică o altă piedică în calea numelor lungi de module.

Singura soluție pentru astfel de probleme este ca autorul pachetului să specifice exact ce module există în pachet. Instrucțiunea `import()` folosește următoarea convenție: dacă codul existent în fișierul `'__init__.py'` definește o listă cu numele `__all__`, aceasta este considerată lista cu modulele ce ar trebui încărcate la execuția unei instrucțiuni `from packet import *`. Este obligația 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 folosește pachetul să execute instrucțiuni de genul `import *`. Fișierul `'Sounds/Effects/__init__.py'` ar trebui să conțină următoarele:

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

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

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

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

În exemplul anterior, modulele `echo` și `surround` sunt importate în spațiul numelor curent deoarece sunt definite în pachetul `Sound.Effects` când instrucțiunea `from ... import` este executată (acest lucru funcționează și când se definește `__all__`).

Nu este foarte eficient să folosiți instrucțiuni de tipul `import *`, pentru că în acest fel codul dumneavoastră va deveni ambiguu, nefiind clar care anume module sunt încărcate. Totuși dacă ați lansat interpretorul în mod interactiv, puteți folosi astfel de instrucțiuni pentru a evita scrierea multor instrucțiuni.

În concluzie nu este nimic greșit în folosirea unei instrucțiuni similare cu `from Packet import submodul`, aceasta fiind și sintaxa recomandată.

## 6.4.2 Referențieri între pachete

Apare deseori necesitatea ca într-un submodule să se folosească cod dintr-un alt submodule. De exemplu modulul `surround` s-ar putea folosi de modulele `echo`. De fapt astfel de referințe sunt atât de întâlnite încât instrucțiunea

`import` examinează întâi pachetul “cel mai cuprinzător”, și apoi caută în lista de directoare. Cu toate acestea modulul `surround` poate folosi mult mai simplu o instrucțiune `import echo` sau `from echo import echofilter`. Dacă modulul ce se dorește importat nu se află în pachetul curent, instrucțiunea `import` caută “mai sus” în ierarhia de module a pachetului.

Atunci când pachetele sunt structurate 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 instrucțiunea: `from Sound.Effects import echo`.

### 6.4.3 Pachete în directoare multiple

Pachetele suporta un atribut special, `__path__`. Acest atribut este inițializat ca o lista care conține numele directorului în care se găsește ‘`__init.py__`’-ul pachetului înainte de executarea codului din fișier. Această variabilă se poate modifica, făcându-și efectul pentru viitoarele căutări de module și subpachete conținute în pachet.

Cât timp această componentă nu se utilizează prea des, se poate folosi pentru extinderea setului de module găsite în pachet.

## Intrări și ieșiri

Există câteva modalități de a prezenta rezultatele unui program. Datele pot fi afișate într-un format care poate fi înțeles de utilizator, sau pot fi scrise într-un fișier pentru a putea fi prelucrate mai târziu. Acest capitol va explica câteva dintre posibilități.

### 7.1 Formatarea elegantă a datelor de ieșire

Până acum am întâlnit două metode de a afișa valori: instrucțiunea `print` și *expresii*. (O a treia metodă este folosind metoda `write()` a obiectelor de tip fișier. Fișierul standard de ieșire este referit ca `'sys.stdout'`.)

Adesea veți dori să aveți mai mult control asupra modului de afișare a valorilor. Există două metode pentru a controla modul de afișare: prima este să modificați singur un șir de caractere, folosind diversele operații existente, iar apoi să îl afișați. Modulul `string` conține câteva operații utile pentru manipularea șirurilor de caractere. O a doua metodă este folosirea operatorului `%`, cu un șir, ca argument stânga. Operatorul `%` interpretează argumentul stânga în același mod ca și șirul de formatare al funcției C `sprintf()` aplicându-l asupra argumentului din dreapta și returnând șirul rezultat în urma ecestei formatări. O singură întrebare rămâne: cum pot fi convertite valorile în șiruri de caractere? Din fericire Python poate converti orice tip de valoare în șir de caractere: fie prin funcția `repr()`, fie scriind valoarea între apostroafe (`"`). Iată câteva exemple:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, ('spam', 'eggs'))'
>>> # reverse quotes are convenient in interactive sessions:
... 'x, y, ('spam', 'eggs')'
'(32.5, 40000, ('spam', 'eggs'))'
```

Iată două modalități de a genera un tabel cu pătratele și 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

```

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

Acest exemplu demonstrează utilizarea funcției `string.rjust()`, care aliniază la dreapta un șir de caractere, într-un câmp a cărui dimensiune este dată de utilizator, introducând spații la stânga șirului. Există și alte funcții similare `string.ljust()`, `string.center()`. Aceste funcții nu afișează nimic, nu fac decât să returneze un alt șir de caractere. Dacă șirul primit este mai lung, aceste funcții 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 șirului. Dacă doriți să trunchiați un șir, puteți oricând să folosiți operațiile de porționare (slicing), ca de exemplu: `string.ljust(x,n)[0:n]`.

Mai există o funcție utilă, care ”umple” cu zero-uri un șir, adăugându-le la stânga șirului original, până când acesta ajunge la o anumită dimensiune. Această funcție 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 formătări în șir, trebuie să se transmită 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 funcționează exact ca în C, și nu necesită decât transmiterea corectă a operandului din dreapta. Nerespectarea acestei reguli va genera o excepție. Specificatorul de format `%s` este mai flexibil, dacă parametrul asociat din partea dreaptă nu este de tip șir de caractere, va fi automat convertit la șir de caractere folosind funcția integrată `str()`. Specificatorii de format `%u` și `%p` din C nu sunt acceptați și de Python.

Dacă aveți un șir lung pe care nu doriți să-l împărțiți în mai multe șiruri, ar fi interesant să vă puteți referi la variabile prin nume în loc de poziție. Acest lucru poate fi realizat folosind modalitatea numevariabilă format, ca în următorul 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 doriți să afișați variabilele predefinite folosind funcția `vars()` care întoarce un dicționar cu toate variabilele locale.

## 7.2 Citirea și scrierea fișierelor

Funcția `open()` are ca rezultat un obiect de tip fișier, și este de cele mai multe ori apelată cu doi parametri: numele de fișier și modul de acces la fișier:

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

Primul argument este un șir de caractere care conține numele fișierului ce urmează să fie deschis. Al doilea argument este tot un șir de caractere ce conține doar câteva caractere ce descriu modul în care fișierul va fi utilizat. Modul poate fi:

- 'r' = fișierul va putea fi numai citit
- 'w' = fișierul va putea fie doar scris (în cazul în care un fișier există deja, acesta va fi suprascris)
- 'a' = fișierul va fi deschis pentru actualizare (toate datele scrise vor fi adăugate la sfârșitul fișierului)
- 'r+' = în fișierul ce va fi deschis se pot executa atât operații de scriere cât și de citire.

Pe Windows și Macintosh adăugarea caracterului 'b', la sfârșitul șirului prin care se specifică modul de acces, indică interpretorului Python să deschidă fișierul în mod binar. Există deci modurile de acces 'rb', 'wb', 'r+b'. Windows face distincție între fișierele de tip text și cele binare: caracterele de sfârșit de linie sunt modificate atunci când se scriu sau se citesc date. Aceste modificări "din spatele scenei" sunt binevenite în cazul fișierelor text, dar nu pot face decât rău în cazul fișierelor binare cum sunt .JPG sau .EXE de exemplu. Aveți deci grijă să folosiți modul binar când lucrați cu astfel de fișiere.

## 7.2.1 Metodele obiectelor fișier

Exemplele acestei secțiuni vor presupune că un obiect 'f' de tip fișier a fost deja creat, deci fișierul a fost deja deschis.

Pentru a citi conținutul unui fișier apălați `f.read(dimensiune)`, care citește o cantitate de date și o returnează ca string. Parametrul dimensiune este opțional. Atunci când acest parametru este omis sau este negativ, întregul fișier va fi citit și returnat ca șir de caractere. Apare o problemă, bineînțeles, dacă memoria mașinii dumneavoastră este mai mică decât dimensiunea fișierului. Dacă parametrul dimensiune este transmis funcției, atunci din fișier vor fi citiți cel mult atâția octeți câți sunt specificați prin acest parametru.

Dacă s-a ajuns la sfârșitul fișierului `f.read()` va returna un șir vid(" ") :

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

Metoda `f.readline()` citește o singură linie din fișier. Un caracter "linie nouă"(newline), "\n", este adăugat la sfârșitul fiecărui șir. Acest caracter este omis dacă este vorba despre ultima linie din fișier și dacă acesta nu se termină cu un caracter "linie nouă". Toate acestea fac rezultatul neclar. Dacă rezultatul este un șir gol, atunci a fost atins sfârșitul fișierului, în timp ce dacă rezultatul este doar caracterul "\n" înseamnă că din fișier 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 fișier, întoarce o listă conținând toate liniile din fișier. Dacă metoda este apelată cu argumentul dimensiune, atunci din fișier sunt citiți atâția octeți câți sunt specificați prin acest parametru și încă atâția octeți câți sunt necesari pentru a completa o linie. Această metodă este folosită pentru citirea eficientă pe linii a fișierelor de dimensiuni mari fără a întâmpina dificultăți 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 `f.write(șir)` scrie conținutul șirului de caractere în fișier, întorcând, ca rezultat, None:

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

Dacă se dorește scrierea a altceva decât șir de caractere, atunci, mai întâi, acesta trebuie convertit în șir :

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

Metoda `f.tell()` are ca rezultat un număr întreg reprezentând poziția cursorului în fișier, poziție măsurată în octeți față de începutul fișierului. Pentru a schimba poziția cursorului folosiți metoda `f.seek(deplasare, referință)`. Noua poziție este calculată în felul următor: cursorul va fi deplasat cu deplasare octeți, față de începutul fișierului dacă referință este 0, față de poziția curentă a cursorului

dacă referință este 1 și față de sfârșitul fișierului dacă referință este 2. Dacă al doilea parametru este omis, valoarea lui implicită va fi 0, deci punctul de referință va fi începutul fișierului:

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Salt la al 5-le byte din fisier
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Salt la al 3-lea byte de la sfarsitul fisierului
>>> f.read(1)
'd'
```

Când terminați lucrul cu un fișier, acesta trebuie închis folosind `f.close()`. După închiderea fișierului orice încercare de operație asupra fișierului va eșua:

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

Obiectele fișier posedă și alte metode, cum ar fi `isatty()` și `truncate()` care sunt mai puțin folosite. (Consultați "Python Library Reference" pentru mai multe detalii).

## 7.2.2 Modulul `pickle`

Șirurile de caractere pot fi citite și scrise foarte ușor dintr-un, respectiv într-un fișier. Cu numerele lucrurile se complică puțin. Ați putea să transformați, înainte de scrierea în fișier, numărul în string, apoi să îl scrieți, iar la citire să îl transformați înapoi în număr. Un astfel de mecanism este complet ineficient.

Pentru aceste situații, și altele mult mai complexe, Python dispune de modulul `pickle` care poate transforma orice obiect Python într-un string. Acest proces se cheamă **pickling**, denumirea procesului invers se numește **unpickling**. Între aceste două proceduri string-ul poate fi salvat într-un fișier, transmis în rețea, etc.

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

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

Iar pentru a încărca un obiect dintr-un fișier:

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

Există și alte metode de a transforma un obiect într-un șir fără a-l salva într-un fișier. Pentru mai multe detalii consultați "Python Library Reference".

Prin procedeele de 'pickling' și '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 și încărcate corect(mai bine zis să se comporte corect în procesul de 'pickling' și apoi 'unpickling').





# Erori și excepții

Până acum erorile au fost doar menționate, dar dacă ați încercat exemplele prezentate probabil ați și întâlnit câteva dintre ele. Există (cel puțin) două categorii de erori: erori de sintaxă și excepții.

## 8.1 Erori de sintaxă

Erorile de sintaxă sunt cel mai des întâlnite, atâta timp cât sunteți 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 și afișează o săgeată în dreptul instrucțiunii care a generat-o. Eroarea este cauzată (sau cel puțin detectată) de instrucțiunea dinaintea săgeții. În exemplul de mai sus eroarea este generată de instrucțiunea `print`, deoarece înaintea acestei instrucțiuni ar trebui să existe un caracter `'`. Numele fișierului și numărul liniei care a generat eroarea sunt afișate, astfel încât dacă eroarea provine dintr-un script să o puteți corecta cât mai comod.

## 8.2 Excepții

Chiar dacă o expresie sau o instrucțiune este corectă din punct de vedere sintactic, aceasta poate genera o eroare în momentul în care este executată. Erorile generate (detectate) în timpul execuției se numesc excepții și nu sunt neapărat fatale. Veți întreba cum puteți evita astfel de erori utilizând limbajul Python.

Majoritatea excepțiilor nu sunt tratate de program și 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 by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Ultima linie a mesajului de eroare indică ce s-a întâmplat. Excepțiile sunt de diferite tipuri, iar tipul excepției este

de asemenea afișat în corpul mesajului de eroare. În exemplul anterior: `ZeroDivisionError`, `NameError` și `TypeError`, desemnează tipul excepției. Șirul afișat este un nume predefinit pentru respectivul tip de excepție. Acest lucru este valabil pentru toate excepțiile predefinite, dar se pot defini și alte tipuri de excepții. Numele excepțiilor standard sunt identificatori predefiniți, nu cuvinte rezervate.

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

În [Python Library Reference](#) sunt listate excepțiile implicite și semnificațiile lor.

## 8.3 Tratarea excepțiilor

Este posibil să scrieți programe care tratează anumite tipuri de excepții. 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, altfel procesul de introducere continuă. Procesul poate fi întrerupt folosind combinația de taste 'CTRL-C'. Dacă utilizatorul folosește această combinație de taste, programul este întrerupt, dar înainte este generată excepția: `KeyboardInterrupt`:

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

Instrucțiunea `try` funcționează în felul următor:

- Întâi sunt executate instrucțiunile din blocul `try` (blocul de instrucțiuni dintre instrucțiunile `try` și `except`).
- Dacă nu este generată nici o excepție, instrucțiunile din blocul `except` nu sunt executate și programul continuă.
- Dacă apare o excepție în timpul execuției instrucțiunilor din blocul `try` restul instrucțiunilor din acest bloc sunt sărite. Dacă tipul excepției este acela pe care îl tratează și blocul `except`, atunci sunt executate instrucțiunile din acest bloc. După ce excepția este tratată, execuția programului continuă cu instrucțiunile ce urmează instrucțiunii `try`.
- Dacă excepția generată nu este prevăzută între excepțiile tratate în blocul `except`, ea este transmisă în afara blocului `try`. Dacă nu este găsit un bloc `except` care să trateze excepția, aceasta va fi o *excepție netratată*. Programul este întrerupt, afișând mesajul : *unhandled exception*.

O instrucțiune `try` poate avea mai multe clauze `except`, implementând astfel mai multe tipuri de 'tratament' pentru mai multe tipuri de excepții. Dintre toate clauzele `except` va fi executată cel mult una. Instrucțiunile unei clauze `except`, tratează numai excepțiile generate în blocul `try` căruia clauza îi este asociată, nu și excepțiile ce pot apare în alte clauze `except`. Clauza din exemplul următor poate trata mai multe tipuri de excepții desemnate printr-o listă închisă între paranteze :

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

Ultima clauză `except` poate fi folosită fără a se specifica ce anume excepție tratează, în acest caz această clauză tratează toate excepțiile netratate de celelalte clauze. Puteți folosi această ultimă precauție, și este indicat să o faceți, pentru a ascunde toate erorile ce pot apărea și pe care nu le-ați anticipat. O puteți de asemenea folosi pentru a afișa un mesaj de eroare și apoi a regenera excepția, 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 in integer"
except:
    print "Eroare neasteptata:", sys.exc_info()[0]
    raise

```

Instrucțiunile `try` dispun și de o clauză opțională: `else`. Atunci când aceasta este folosită, ea trebuie să fie ultima dintre clauze. Instrucțiunile acestei clauze vor fi executate atunci când, blocul `try` nu generează nici o excepție. 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 de preferat adăugării unor linii suplimentare de cod la sfârșitul blocului `try`, pentru că în acest fel se evită detectarea unei excepții care nu a fost generată de instrucțiunile blocului `try`.

O excepție poate avea asociată și o valoare, un argument al excepției. Prezența argumentului și tipul acestuia depinde de tipul excepției.

Clauza `except` poate accepta o variabilă după numele excepției, sau o listă de nume de excepții. Variabila este legată de instanța excepției, cu argumentele salvate în `instance.args`. Prin convenție instanța excepției definește `__getitem__` și `__str__` și argumentele pot fi accesate, sau listate, direct, fără a mai referi `.args`.

```

>>> try:
...     raise Exception('spam', 'eggs')
... except Exception, inst:
...     print type(inst)          # instanta exceptiei
...     print inst.args           # argumentele stocate in .args
...     print inst                # __str__ permite args sa fie listate direct
...     x, y = inst               # __getitem__ permite args sa fie despachetate direct
...     print 'x =', x
...     print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

Dacă o excepție are un argument, acesta va fi afișat în ultima parte ("detaliu") a mesajului de excepție netratată. Într-un bloc `try` nu sunt tratate numai excepțiile generate imediat în cadrul blocului, ci și dacă acestea apar în funcțiilor apelate (chiar și indirect) în bloc. De exemplu:

```
>>> 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 excepțiilor

Instrucțiunea `raise` îi permite programatorului să genereze o anumită excepție :

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

Primul parametru reprezintă tipul excepției, iar al doilea este opțional și reprezintă un eventual argument.

Dacă se vrea să se știe când o excepție a fost semnalată, dar nu se intenționează tratarea ei, o formă mai simplificată a instrucțiunii `raise` permite reapariția excepției :

```
>>> 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 Excepții definite de utilizator

Un programator își poate crea propriile excepții prin crearea unei clase de excepții noi. În mod obișnuit excepțiile pot fi derivate din clasa `Exception` atât în mod direct cât și 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 excepții pot fi proiectate ca oricare alta clasă, dar în 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 obișnuiește să se creeze o clasă de bază a excepțiilor definite de acest modul și subclase care să creeze clase specifice pentru diferite condiții de eroare :

```

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

class InputError(Error):
    """Excetie generata pentru eroare de intrare.

    Atribute:
        expression -- expresia de intrare in care apare eroarea
        message -- explicarea erorii
    """

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

class TransitionError(Error):
    """Generata cand o operatie asteapta o modificare
    de stare care nu este permisa

    Attributes:
        previous -- starea la inceputul tranzitiei
        next -- noua stare care este de asteptat sa apara
        message -- explicatie privind imposibilitatea tranzitiei
    """

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

```

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

Multe module își definesc propriile excepții pentru a raporta erorile ce pot apărea în funcțiile definite de ele. Mai multe informații despre clase veți găsi în capitolul dedicat acestora.

## 8.6 Definirea acțiunilor de curățare

Instrucțiunea `try` poate avea și o altă clauză prin care se pot defini acțiuni de curățare care vor fi executate în orice circumstanță. 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 excepție în blocul `try`. Când apare o excepție, aceasta este regenerată după ce sunt executate instrucțiunile clauzei *finally*. Clauza *finally* este de asemenea executată “la ieșire”, chiar și atunci când a fost apelată o instrucțiune `break` sau `return`.

Codul scris în clauza *finally* este util pentru eliberarea resurselor externe (cum ar fi fișiere sau conectări la rețea), fara a mai vedea daca utilizarea resurse s-a încheiat cu succes.

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

# Clase

Mecanismul de clase al limbajului Python face posibilă adăugarea de noi clase cu un minim de efort. Acest mecanism de clase este o combinație între mecanismele din C++ și Modula-3. La fel ca și pentru module, Python nu creează o barieră absolută între definiție și utilizator, ci se bazează mai mult pe “politețea” utilizatorului de a nu “pătrunde în definiție”. Cele mai importante facilități ale claselor sunt în continuare disponibile : mecanismul de moștenire permite clase de bază multiple, o clasă derivată poate suprascrie orice metodă a clasei/claselor de bază, o metodă poate apela o metodă a clasei de bază cu același nume. Obiectele au un număr arbitrar de date private.

În terminologie C++, toate componentele unei clase (inclusiv datele membre) sunt *publice*, și toate funcțiile membre sunt *virtuale*. Nu există constructori sau destructori specializați. La fel ca în Modula-3, nu există scurtături pentru a accesa membrii unui obiect din cadrul unei metode a acelui obiect: metodele sunt declarate cu un prim argument explicit reprezentând obiectul, care este apoi transmis automat în momentul apelării. Ca și în Smalltalk clasele însăși sunt în sine obiecte ; în sens mai larg: în Python toate tipurile de date sunt obiecte. Astfel sunt posibile importurile și redenumirile. Spre deosebire de C++ și Modula-3, tipurile predefinite pot fi utilizate drept clase de bază de către utilizator. De asemenea, la fel ca în C++, dar spre deosebire de Modula-3, majoritatea operatorilor cu sintaxă specială (de exemplu: operatorii aritmetici) pot fi redefiniți în instanțierile claselor.

## 9.1 Câteva cuvinte despre terminologie

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

Va trebui de asemenea să avertizăm cititorii asupra unei capcane de terminologie : cuvântul : în Python cuvântul “*obiect*” nu înseamnă obligatoriu o instanțiere a unei clase. Ca și în C++ și Modula-3, dar nu și ca în SmallTalk, în Python unele tipuri de date nu sunt neapărat clase: numerele întregi și listele, dar și alte tipuri de date mai “exotice”, ca fișierele nu sunt clase. Toate tipurile de date din Python au, semantic, ceva comun, care este mai ușor de explicat dacă ne referim la ele folosind cuvântul “obiect”.

Obiectele pot avea individualitate, sau pot avea mai multe nume(care pot servi mai multor scopuri) care să desemneze același obiect. Această procedură se numește, în alte limbaje, “aliasing”. 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, șiruri, perechi). În orice caz aliasing are și un efect (intenționat) asupra sintaxei Python care privește obiectele alterabile (liste, dicționare, etc.) și obiectele ce există în afara contextului în care se află programul (fișiere, ferestre. etc.). Puteți utiliza tehnica aliasing, pentru a beneficia de puterea oferită în alte limbaje, de pointeri, întrucât puteți considera că alias-urile funcționează, 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 funcție primește ca parametru un obiect, pe care îl modifică, apelantul va putea observa modificările făcute asupra obiectului și nu asupra reprezentării locale. Acestea elimină necesitatea folosirii a două mecanisme de transmitere a parametrilor, ca în Pascal.

## 9.2 Domenii de vizibilitate (Scopes) și domenii de definiție a numelor(Name Spaces)

Definițiile claselor manipulează într-un mod foarte interesant domeniile de definiție a numelor, și o foarte bună înțelegere a modului de funcționare a domeniilor de vizibilitate și de definiție a numelor este foarte importantă.

Înțelegerea acestor mecanisme este foarte importantă și se va dovedi foarte utilă oricărui programator avansat Python.

Să începem cu câteva definiții.

Un *domeniu de definiții a numelor* este o “hartă” de legături între nume și obiecte. În momentul de față aceste domenii sunt implementate folosind dicționare, 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 funcții, etc. Într-un anumit sens atributele unui anume obiect formează un domeniu de definiție. Un lucru foarte important de știut 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 funcție cu aceeași denumire fără a se crea 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, referințele la numele definite într-un modul sunt de fapt referințe la atribute ale modulului: în expresia `modul.functie`, `modul` este modulul iar `functie` este un atribut al modulului.

Numele globale ale modulului și atributele modulului se află în același domeniu de definiție al numelor.(Există o singură excepție. Modulele au un atribut nemodificabil (read-only) “secret”: `__dict__`, în care se află domeniul de definiție al numelor ce aparține 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: se poate folosi o secvență de genul: `modul.răspuns = 42`. Atributele modificabile pot fi de asemenea șterse folosind operatorul `del`. De exemplu `del modul.răspuns` va elimina atributul `răspuns` din obiectul denumit `modul`.

Domeniile de definiție a numelor sunt create la momente de timp diferite și au perioade de viață diferite. Domeniul care conține numele predefinite este creat odată cu pornirea interpretorului Python și nu este șters niciodată. Domeniul numelor globale definite într-un modul este creat atunci când este încărcată definiția modulului și este distrus, de obicei, tot în momentul în care interpretorul este închis. Instrucțiunile 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 definiție a numelor pentru o funcție este creat odată cu apelul funcției și este șters odată cu terminarea funcției, fie prin `return`, fie dacă acesta generează o excepție netratată. Bineînțeles că apeluri recursive generează mai multe domenii de definiție.

Un *domeniu de vizibilitate* este o regiune a unui program Python în care un anumit domeniu de definiție a numelor este accesibil. Accesibil înseamnă că o anumită referință la un nume va genera o căutare în domeniul de definiție al acelui modul, despre care spunem că este accesibil.

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

De obicei, “cel mai adânc” domeniu de vizibilitate este cel al funcției care este executată. În exteriorul unei funcții, acest domeniu este identic cu al doilea domeniu (cel din mijloc), în care se află accesibile definițiile modulului.

Domeniul de vizibilitate global al unei funcții definite într-un modul este domeniul de definiții al respectivului modul, indiferent de unde, sau cu ce alias este apelată respectiva funcție. Pe de altă parte căutarea unui anumit nume este realizată dinamic, în timpul execuției. Cu toate acestea definiția limbajului evoluează către o “rezoluție” a numelor statică, în timpul compilării. În concluzie nu vă bazați prea tare pe varianta dinamică. Deja variabilele locale sunt determinate static.

O caracteristică a limbajului Python este aceea că atribuirile au loc în primul domeniu de vizibilitate(“cel mai de



jos” sau “cel mai adânc”)<sup>1</sup>. Atribuirile nu realizează o copiere a datelor, ci leagă nume de obiecte. Acest lucru este valabil și pentru ștergeri. Instrucțiunea `del x` șterge legătura lui `x` din domeniul local de definiție. De fapt toate operațiunile care introduc nume noi au impact asupra domeniului de definiții local: instrucțiunile `import`, sau definiția unei funcții, introduc numele modulului importat, respectiv numele funcției, în domeniul de definiții local. Dacă este folosită instrucțiunea `global` se indică interpretorului ca următorul nume să fie introdus în domeniul global.

## 9.3 O primă privire asupra claselor

Noțiunea de clasă aduce cu sine și câteva alte noțiuni noi de sintaxă, trei tipuri noi de obiecte, și câteva noțiuni noi de semantică.

### 9.3.1 Sintaxa de definire a unei clase

Cea mai simplă definiție a unei clase arată astfel:

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

Definițiile de clase, la fel ca definițiile de funcții, sunt valabile de abia după ce sunt executate. Puteți insera o definiție de clasă în cadrul unei instrucțiuni `if`, și clasa nu va exista decât dacă condiția instrucțiunii `if` va fi adevărată.

În practică, instrucțiunile din interiorul unei clase vor fi definiții de funcții, dar sunt permise și alte instrucțiuni care pot fi uneori foarte utile. Definițiile de funcții din cadrul unei clase sunt un pic deosebite față de ce cunoașteți până acum, dar despre asta vom vorbi mai târziu.

Atunci când este introdusă definiția unei clase noi, un nou domeniu de definiții a numelor este creat și folosit ca domeniu de vizibilitate local, deci toate atribuirile vor afecta noul domeniu de definiții creat. Definițiile de funcții vor introduce numele acelor funcții tot în acest domeniu.

Când definiția unei clase se termină normal, fără erori, este creat un obiect nou de tipul `class`. Pe scurt, acest obiect “conține” noul domeniu de definiții creat odată cu definiția clasei. În următoarea secțiune vom discuta mai mult despre obiectele de tip `class`. Domeniul de vizibilitate existent înaintea definiției clasei este reactivat, și obiectul `class` este introdus în respectivul domeniu de definiții sub numele dat de numele specificat în definiția clasei (în exemplul de mai sus: `ClassName`).

### 9.3.2 Obiecte clasă (class)

Această categorie de obiecte acceptă două categorii de operații: referențieri de attribute și instanțieri.

Referențierile de attribute folosesc sintaxa standard din Python: `obiect.attr`. Attribute valide sunt toate numele existente în domeniul de definiții în momentul creării obiectului `class`. Așa că, dacă definiția clasei arată astfel:

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

---

<sup>1</sup> N.T. termenul original este `innermost`

atunci `MyClass.i` și `MyClass.f` sunt referințe la atribute valabile. Unul dintre atribute este un întreg (i), iar celălalt o metodă(f). Se pot realiza atribuiri cu atributele unei clase, deci puteți modifica valoarea atributului `MyClass.i` printr-o atribuire. `__doc__` este de asemenea un atribut al clasei, care are valoarea : “Un exemplu simplu de clasă”.

Instanțierea claselor folosește notația de la funcții. Putem să ne imaginăm că obiectul clasă este o funcție fără parametrii care întoarce, ca rezultat, o nouă instanță pentru respectiva clasă. Exemplul următor:

```
x = MyClass()
```

crează o nouă instanță a clasei și atribuie acest obiect variabilei locale `x`.

Operația de instanțiere crează un obiect vid. Multe clase preferă să creeze un obiect într-o stare inițială. Pentru a realiza acest lucru clasa 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__()`, la instanțierea ei se apelează automat metoda `__init__()`. În acest fel se obține o instanță inițializată a clasei :

```
x = MyClass()
```

Metoda `__init__()` poate avea și argumente. În acest caz argumentele primite la instanțierea 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 puteți gândi la metoda `__init__()` ca la un constructor, deși nu este un constructor în toată puterea cuvântului.

### 9.3.3 Obiecte instanțiate

Ce putem face cu instanțele? Singurele operațiuni acceptate sunt operațiunile cu atribute. Există două tipuri de nume de atribute valide : proprietăți și metode.

Prima categorie sunt *proprietățile*, sau atribute de tip dată. Această categorie corespunde “variabilelor instanței” în SmallTalk, și “membrilor dată” în C++, create automat atunci când le sunt atribuite valori. Dacă `x` este o instanță a clasei `MyClass` următoarea porțiune de cod va afișa 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 funcție ce aparține unui anumit obiect.(În Python termenul de metodă nu desemnează doar o funcție ce aparține numai instanței unei clase, adică unui obiect clasă; și 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 definiție toate atributele de tip funcție ale unei clase devin metode pentru toate instanțele respectivei clase. În exemplul nostru `x.f` este o metodă valabilă din moment ce `MyClass.f` este o funcție. În mod contrar este evident că `x.i` nu este o metodă, pentru că `MyClass.i` nu este o funcție. Există o diferență între `MyClass.f` și `x.f`: `MyClass.f` este un obiect funcție iar `x.f` este un obiect metodă (una este funcție, a doua metodă).

### 9.3.4 Obiecte metodă

În mod uzual o metodă este apelată imediat:

```
x.f()
```

În exemplul nostru aceasta va returna șirul 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 afișa `Hello World!` la infinit.

Ce se întâmplă exact atunci când este apelată o metodă? Ați observat că `x.f()` a fost apelată fără nici un argument, chiar dacă în definiția funcției, din corpul clasei, era specificat un argument.

Ce s-a întâmplat cu argumentul? Cu siguranță Python generează o excepție atunci când o funcție care necesită un argument este apelată fără el, deși s-ar putea nici să nu fie utilizat.

Probabil că ați 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 funcția corespondentă inserând la începutul listei de `n` argumente obiectul a cărui metodă se dorește apelată.

Dacă nu ați înțeles încă cum funcționează metodele, o privire asupra implementării probabil că ar clarifica lucrurile. Atunci când se face o referire la un atribut, care nu este un atribut dată, al unei instanțe este căutat în clasa din care face parte. Dacă numele reprezintă un atribut existent în respectiva clasă (un obiect de tip funcție), este creat un obiect metodă “împachetând” obiectul instanță și obiectul funcție (tocmai găsit). În acest fel se creează un obiect metodă. Atunci când un obiect metodă este apelat cu o listă de argumente, el este despachetat din nou, o nouă listă de argumente este construită din obiectul instanță și lista de argumente originală. În cele din urmă obiectul funcție este apelat cu noua listă de argumente.

## 9.4 Alte observații

Atributele de tip dată suprascriu atributele metodă cu același nume. Pentru a evita conflicte de nume, ce pot cauza erori greu de găsit în programe mari, este bine să folosiți o convenție pentru denumiri. O posibilă convenție ar fi scrierea numelor de metode cu literă inițială mare, prefixarea numelor de atribute dată cu un șir de caractere mic și unic, (poate fi un underscore), sau pentru metode folosiți verbe, iar pentru proprietăți substantive.

Atributele dată pot fi accesate de metode la fel de bine ca și 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 convenție. (Pe de altă parte, implementarea limbajului Python, realizată 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 (proprietățile) cu atenție - orice modificare directă a acestora putând duce la o “incoerență” 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(și aici existența unei convenții vă poate scuti de multe neplăceri).

Nu există nici o scurtătură pentru referirea atributelor din interiorul unei metode. Acest lucru clarifică codul unei metode, nemaexistând posibilitatea confuziei între variabilele locale și variabilele instanței.

În mod convențional, primul argument al unei metode are numele: `self`. Aceasta este numai o convenție, numele `self` neavând nici o semnificație specială în Python. Folosind această convenție codul dumneavoastră va fi mai ușor de citit de către programatori.

Orice obiect funcție, ca atribut al unei clase, definește o metodă asociată pentru instanțele unei clase. Nu este neapărat necesar ca definiția unei funcții să se afle în cadrul definiției clasei. În acest sens se poate atribui un obiect funcție unei variabile locale. De exemplu:

```
# Functie definita inafara 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`, și `h` sunt toate atribute ale clasei `C`, și sunt atribute funcției, și în mod consecvent sunt metode pentru toate instanțele 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 același mod ca funcțiile obișnuite. Domeniul global de vizibilitate asociat unei metode permite accesul la domeniul de definiție a numelor asociat modulului în care se află definită clasa căreia îi aparține metoda.

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

## 9.5 Moștenirea

Nu putem spune despre un limbaj că este orientat pe obiect, dacă sistemul de clase nu pune la dispoziție mecanismul de moștenire. Sintaxa pentru moștenire (adică pentru crearea definiției 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 definiția clasei derivate să fie corectă. În loc de un nume de clasă puteți 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 și 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 și în clasa de bază. Regula se aplică recursiv dacă clasa de bază este la rândul ei o clasă derivată.

Instanțierea unei clase se face la fel ca până acum. Dacă este apelată o metodă a unei instanțe, numele metodei este căutat întâi în definiția clasei derivate, apoi în definiția clasei de bază, și așa mai departe până când este găsit un obiect funcție corespunzător în lanțul de moșteniri.

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 aceluiași 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 direct o metodă a clasei de bază: `ClasădeBază.metodă(self.argumente)`.

Acest lucru poate fi foarte util chiar și utilizatorilor (clienților) unei clase.

### 9.5.1 Moștenirea multiplă

Python pune la dispoziția programatorilor și o formă de moștenire multiplă. O definiție 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 definiția clasei `Base1` și în toate clasele de la care aceasta moștenește, apoi este căutat în `Base2` și în `Base3` după aceleași reguli. Putem rezuma această regulă în felul următor: “întâi în adâncime, și de la stânga la dreapta”. (Avantajele și 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 moștenire multiplă va crea mari probleme de întreținere, deoarece în Python conflictele ce pot apărea între nume sunt evitate numai prin convenții. O problemă cunoscută apare atunci când o clasă moștenește de la două clase care la rândul lor sunt derivate dintr-o singură clasă. Deși 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 situației.

## 9.6 Variabile private

Există un suport limitat pentru identificatorii clselor private. Orice identificator de forma `__spam` (cel puțin două underscore înainte, cel mult un underscore după) este acum textual înlocuit cu `_clasă__spam`, unde `clasă` este numele curent al clasei cu underscore-urile din față șuprimate. Mutilarea este realizată fara o analiza a poziției sintactice a identificatorului, așa încât putând fi folosită la definirea unei instanțieri a unei clase private și a variabilelor și metodelor clasei ca globale, și la memorarea instanțierii variabilelor private în această clasă la instanțierea *altei* clase. O truncare a numelui poate apărea când acesta este mai lung de 255 de caractere. În afara claselor, sau când numele clasei este alcătuit numai din underscore-uri, nu se face eliminarea underscore-urilor.

Acest mecanism pune la dispoziția programatorului o metodă foarte facilă de a crea atribute private, fără a-și face griji că aceste atribute pot provoca conflicte de nume. Toate acestea sunt realizate în Python pentru a preveni eventualele erori, totuși, 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 același nume cu clasa de bază poate folosi variabilele private ale clasei de bază).

Observați cum codul ce este trimis instrucțiunilor `exec`, `eval()` sau `evalfile()` nu ia în considerare numele clasei care realizează invocările, acest efect fiind similar celui realizat prin instrucțiuni `global`, efect ce nu poate fi obținut pentru cod compilat. Aceeași restricție se aplică și pentru `getattr()`, `setattr()` și `delattr()`, de asemenea și pentru accesarea directă a dicționarului `__dict__`.

Iată un exemplu de clase care implementează propriile metode `__getattr__()` și `__setattr__()`, și 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 puteți folosi clasele. O definire a unei clase vide se poate face astfel :

```
class Employee:
    pass

john = Employee() # Se creaza o inregistrare vida 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 aveți o funcție care prelucrează anumite date dintr-un obiect fișier, puteți să definiți o clasă cu metodele `read()` și `readline()` care să citească informații dintr-un șir de caractere

în loc de un fișier.

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

## 9.8 Excepțiile pot fi clase

Excepțiile definite de utilizator pot fi atât obiecte de tip `string`, cât și clase. Folosind clasele pentru a defini excepții, puteți crea o întreagă ierarhie de excepții care poate fi foarte ușor extinsă.

Există o altă formă a instrucțiunii `raise`:

```
raise Class, instance

raise instance
```

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

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

O clauză `except` poate afișa atât excepții de tip clasă, cât și de tip șir de caractere. Următorul exemplu va afișa 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 afișat B, B, B; deoarece clasa B este clasa din care sunt derivate clasele C și D.

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

## 9.9 Iteratori

Pâna acum ați observat că majoritatea obiectelor container se pot baleia utilizând instrucțiunea `for` :

```

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

```

Acest mod de accesare este clar, concis și comod. Folosirea iteratorilor este larg răspândită în proiectarea Python-ului. Prin modul său de implementare instrucțiunea `for` apelează `iter()` pentru obiectul container. Funcția returnează un obiect iterator care definește metoda `next()` prin care se accesează doar o singură dată fiecare element al containerului. Când toate elementele au fost epuizate, `next()` setează excepția `StopIteration`, care comunică buclei `for` să se oprească. Modul de funcționare este ilustrat de următorul exemplu :

```

>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration

```

S-a văzut mecanismul din spatele procesului de iterare. Este ușor de a implementa un comportament iterativ claselor proiectate de către utilizator. Se definește o metodă `__iter__()`. Această metodă returnează un obiect ce va avea o metodă `next()`. Dacă clasa definește `next()`, atunci `__iter__()` va returna doar `self` :

```

class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s

```



## 9.10 Generatori

Generatorii sunt instrumente simple și puternice pentru crearea iteratorilor. Generatorii se scriu ca și funcțiile obișnuite, dar se utilizează instrucțiunea `yield` când se dorește o returnare de date. De fiecare dată se apelează `next()`, generatorul reluând din locul de întrerupere (el memorează toate valorile datelor și ce instrucțiune a fost executată ultima). Exemplul următor prezintă cât de simplu se poate crea un generator :

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Tot ce se poate face cu generatori se poate face și cu clase bazate pe iteratori, descrise în secțiunea anterioară. Ce face ca generatorul să fie mai compact este faptul că metodele `__iter__` și `next()` sunt create automat.

O altă caracteristică importantă este aceea că între apeluri sunt salvate variabilele locale și starea execuției. Acest lucru face funcția mai ușor de scris și mai clară decât abordarea cu variabile de clasă precum `self.index` și `self.data`.

În plus, la salvarea automată a variabilelor și a stării programului, când generatorul își încheie lucrul, setează automat `StopIteration`. Toate aceste caracteristici dau posibilitatea de a scrie iteratori la fel de ușor ca și o funcție obișnuită.

## 9.11 Expresii generator

Unii generatori simpli se pot scrie succint ca expresii folosind o sintaxă similară cu cea a creării listelor obișnuite, dar cu paranteze în locul acoladelor. Aceste expresii sunt create pentru situațiile în care generatorii sunt ceruți ca atare în corpul unei funcții. Expresiile generator sunt mult mai compacte, dar și mult mai puțin cuprinzătoare decât definirea completă a generatorilor și tind să fie mai ușor de memorat decât datele de tip listă. De exemplu :

```
>>> sum(i*i for i in range(10))                # suma de patrate
285
>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))
260
>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))
>>> unique_words = set(word for line in page for word in line.split())
>>> valedictorian = max((student.gpa, student.name) for student in graduates)
>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```



# Pe scurt despre Standard Library - partea I

## 10.1 Interfața cu sistemul de operare

Modulul `os` conține o serie de funcții pentru interfațarea cu sistemul de operare :

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()                # Returneaza directorul de lucru curent
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Asigurați-vă că folosiți `'import os'` în loc de `'from os import *'`. Aceasta va împiedica `os.open()` să estompeze efectul funcției predefinite `open()`, care lucrează într-un mod mult diferit. Funcțiile predefinite `dir()` și `help()` sunt foarte utile ca un ajutor interactiv pentru lucrul cu module de dimensiuni mari, cum este cazul modulului `os` :

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Pentru administrarea fișierelor curente și a directoarelor de lucru, modulul `shutil` furnizează o interfață de nivel înalt, ușor de folosit :

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

## 10.2 Fișierele Wildcard (asterisc)

Modulul `glob` oferă o funcție pentru realizarea listei cu fișierele căutate într-un director folosind asterisc în locul numelui :

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

## 10.3 Argumentele în linia de comandă

Scripturile obișnuite reclamă adesea procesarea liniilor de comandă cu argumente. Aceste argumente sunt depozitate, sub formă de listă, în atributele *argv* al modulului *sys*. Spre exemplu, în urma execuției următoarei linii de comandă `'Python demo.py one two three'` va rezulta :

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Modulul *getopt* prelucrează *sys.argv* aplicând regula funcției UNIX *getopt()*. O prelucrare mai puternică și mai flexibilă a liniei de comandă este oferită de modulul *optparse*.

## 10.4 Redirectarea semnalării erorilor și terminarea execuției programului

Modulul *sys* conține și atribute pentru *stdin*, *stdout* și *stderr*. Cel din urmă este utilizat pentru generarea de avertismente și mesaje de eroare, pentru a le face vizibile chiar și când *stdout* este redirectat :

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

Modalitatea cea mai directă de a încheia un scrip este aceea de a folosi `'sys.exit()'`.

## 10.5 Unificarea tratării șirurilor

Modulul *re* oferă ca instrumente expresii standard pentru procesarea de nivel avansat a șirurilor. Pentru potriviri complexe și pentru lucrul cu șirurile, sunt puse la dispoziția utilizatorului expresii care oferă soluții optimizate și rapide :

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Când impune doar capacități reduse, atunci este de preferat lucrul cu metodele specifice șirurilor, pentru că sunt ușor de citit și de depanat :

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6 Module matematice

Modulul `math` oferă acces la funcțiile din structura bibliotecii C pentru calculul în virgulă mobilă :

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

Modulul `random` are unelte pentru a face selecții aleatorii :

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)          # exemplificare fara inlocuire
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()                        # flotent obtinut aleatoriu
0.17970987693706186
>>> random.randrange(6)                    # intreg aleatoriu obtinut cu range(6)
4
```

## 10.7 Accesul la Internet

Sunt o mulțime de module pentru accesarea internetului și pentru procesarea protocoalelor de Internet. Două dintre cele mai simple sunt `urllib2`, pentru obținerea datelor de la *url*-uri, și `smtplib`, pentru trimiterea mesajelor prin poștă electronică :

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line:          # look for Eastern Standard Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

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

Beware the Ides of March.
""")
>>> server.quit()
```

## 10.8 Data și timpul

Modulul `datetime` furnizează clase pentru prelucrarea datei și a timpului atât într-o manieră simplistă, cât și într-una mai elaborată. Modulul implementează și aritmetica datei și a timpului. O atenție deosebită se acordă accesării eficiente a membrilor, pentru formatarea lor la afișare și pentru prelucrare. De asemenea sunt implementate și obiecte care sunt legate de ora locală.

```

# datele sunt usor de construit si formatat
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# datele suporta artmetica de tip calendar
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368

```

## 10.9 Compresarea datelor

Formatele de arhivare și compresare obișnuite sunt accesate direct prin includerea modulelor `zlib`, `gzip`, `bz2`, `zipfile` și `tarfile`.

```

>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(t)
-1438085031

```

## 10.10 Măsurarea performanței

Unii utilizatori Python sunt interesați de cunoașterea diferențelor de performanță între diferite abordări ale aceluiași probleme. Python oferă instrumente de măsură, care răspund rapid cererilor.

Există tentația folosirii împachetării și a despachetării perechilor (tuple) în locul obișnuitei schimbări între argumentelor. Modulul `timeit` evidențiază imediat un mic avantaj de performanță:

```

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

```

În contrast cu `timeit`, care operează pe coduri de foarte mici dimensiuni, modulele `profile` și `pstats` au instrumente pentru identificarea secțiunii de timp critic pentru aplicații de mari dimensiuni.

## 10.11 Controlul calității programului

O abordare a programării de performanță este aceea când pentru fiecare funcție se știu, în faza de proiectare, teste ce vor fi rulate pe perioada întregului proces de dezvoltare a aplicației.

Modulul `doctest` oferă un instrument de scanare a unui modul și de verificare a bunei funcționări a testului implantat în *docstring*-ul programului. Implantarea testului în *docstring* este ușoară. În *docstring* se va înscrie o apelare a funcției împreună cu rezultatul execuției. Această implementare oferă utilizatorului un exemplu de utilizare a funcției și permite modulului *doctest* să se verifice conformitatea codului cu descrierea din documentație :

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

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

import doctest
doctest.testmod()    # verificarea automata a testului din documentatia functiei
```

Cu modulul `unittest` nu se lucrează mai ușor decât cu modulul `doctest`, dar el oferă posibilitatea scierii unui program într-un fișier cu un set complet de teste.

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

unittest.main() # Calling from the command line invokes all tests
```

## 10.12 Python oferit "la cheie"

Limbajul Python este un limbaj oferit "la cheie", lucru dovedit de complexitatea și robustețea întregii distribuții. De exemplu :

- Modulele `xmlrpclib` și `SimpleXMLRPCServer` fac implementări de proceduri apelabile de la distanță de către aproape orice aplicație. În ciuda numelui nu este necesară cunoașterea sau utilizarea XML.
- Pachetul `email` este o bibliotecă pentru prelucrarea mesajelor de email, incluzând și mesajele tipice MIM și RFC 2822. Spre deosebire de `smtp` și `pop`, care, de fapt, transmit și recepționează mesajele, pachetul `email` oferă un set complet de instrumente necesare la constituirea și decodarea structurilor complexe ale mesajelor (inclusiv atașare de fișiere - *attachments*) și pentru implementarea protocoalelor de codare Internet și a headerelor.
- Pachetele `xml.dom` și `xml.sax` oferă un suport puternic pentru analizarea unui text scris XML, un format foarte răspândit. În același fel modulul `csv` suportă scrierea și citirea într-un format de bază de date comun. Împreună aceste module și pachete simplifică într-un mod elegant schimbarea de date între aplicațiile Python și alte instrumente.

- Internaționalizarea este asigurată de un număr de module ce înglobează pachetele `gettext`, `locale` și `codecs`.



## Pe scurt despre Standard Library - partea II

În partea a doua a scurtei prezentări a Standard Library sunt analizate modulele necesare unei programări profesionale. Aceste module apar arareori în scripturile mici.

### 11.1 Formatarea datelor de ieșire

Modulul `repr` furnizează o versiune a funcției `repr()` pentru afișarea prescurtată a unor containere de lungime mare :

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

Modulul `pprint` execută un control mai sofisticat, atât asupra listării obiectelor predefinite, cât și a celor definite de utilizator, într-un mod care poate fi ușor citit de utilizator. Când reprezentarea este mai lungă decât o linie, *"pretty printer"* forțează sfârșitul liniei și face alinierea, pentru o mai mare claritate în afișarea structurii de date :

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

Modulul `textwrap` formatează paragrafele textului pentru a se putea încadra într-o anumită dimensiune impusă de programator pentru fereastra de afișare a datelor :

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

Modulul `locale` accesează o bază de date cu formate de reprezentare specifice diferitelor zone culturale ale lumii. Prin gruparea formatelor specifice zonelor se pot afișa numerele având caractere specifice de separare a cifrelor, precum și semnul monetar specific :

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

## 11.2 Șirurile șablon (Templating)

Modulul `string` conține o clasă `Template` flexibilă, cu o sintaxă simplificată, utilă în editare. Aceasta permite utilizatorului să-și personalizeze aplicațiile.

Formatul folosește cuvinte șablon alcătuite dintr-un delimitator (caracterul `$`) urmat de un identificator recunoscut de Python (șir de caractere alfanumerice și underscore). Încadrând șablonul între acolade se permite continuarea acestuia cu un șir de caractere alfanumerice, care să nu fie separate prin spații. Caracterul `$` se reprezintă scriind `$$` :

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

Metoda `substitute` setează `KeyError` când un șablon nu este găsit într-un dicționar, sau într-un argument de tip cuvânt cheie.

Pentru aplicațiile de tip email utilizatorul poate omite completarea unor date. De aceea se recomandă utilizare metodei `safe_substitute`, ea lasând nemodificat șablonul cu data corespunzătoare lipsă :

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Pentru șirurile șablon utilizatorul poate alege un anumit delimitator. De exemplu se poate alege drept delimitator caracterul % :

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f

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

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

O altă aplicabilitate a șirurilor șablon se găsește în prelucrarea datelor de ieșire scrise în diferite formate. Aceasta se realizează înlocuind șabloanele definite de utilizator pentru fisierele XML, rapoartele text, sau rapoartele web scrise în HTML.

## 11.3 Lucrul cu reprezentări binare ale datelor

Modulul `struct` are funcțiile `pack()` și `unpack()` pentru lucrul cu formate de înregistrare binară a datelor cu lungime variabilă. Exemplul următor prezintă cum se pot căuta informații prin header-ul unui fișier ZIP (cu cod de împachetare "H" și "L", reprezentând numere fără semn de doi, respectiv, patru octeți) :

```
import struct
data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):
    start += 14
    fields = struct.unpack('LLLHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields
    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size
    start += extra_size + comp_size # sare la urmatorul header
```

## 11.4 Multi-threading

Threading este o tehnică prin care se realizează decuplarea proceselor care nu sunt dependente secvențial unul de celălalt. Această tehnică se poate folosi pentru îmbunătățirea răspunsului aplicațiilor care primesc date de la utilizator, atunci când alte procese rulează în background. Un astfel de caz este acela în care un proces de intrare/ieșire rulează în paralel cu un proces de calcul.

Următorul program prezintă cum modulul de nivel înalt `threading` poate executa procese în background, în timp ce programul principal continuă să ruleze :

```

import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'

```

Principala provocare în proiectarea aplicațiilor multi-threading este aceea de a coordona thread-urile care partajează aceleași date, sau aceleași resurse. La sârșit, modulele *thread* generează un număr de primitive de sincronizare precum : variabile de blocare, de stare, indicatori de tip semafor.

## 11.5 Conectivitatea (Logging)

Modulul logging oferă un sistem de conectare flexibil și cu toate caracteristicile specifice unui astfel de sistem. Mesajele de conectare sunt transmise către un fișier, sau către `sys.stderr` :

```

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

```

Acestea transmit următoarele :

```

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

```

Implicit mesajele cu informații și cele de depanare sunt suprimate și ieșirea este direcționată către *stderr*. Alte opțiuni de ieșire includ rutarea mesajelor prin email, datagrame, socket, sau server HTML. Folosind filtre noi se pot selecta diferite rute funcție de prioritatea mesajului : `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.

Sistemul de conectare (login) poate fi configurat direct de Python, dar poate fi și preluat dintr-un fișier de configurare editat de utilizator, pentru a particulariza conectarea fără a altera setarea mediului.

## 11.6 Legături slabe la memorie

Pentru fiecare obiect, Python realizează o legătură a acestuia cu memorie. Legătura poate fi *puternică* (permanentă), sau *slabă* (de scurtă durată). Aceste legături sunt administrate în mod automat, prin contorizarea legăturilor obiectelor și prin folosirea "*coșului de gunoi*" pentru eliminarea ciclurilor. Memoria este eliberată imediat ce ultima referire la ea a fost eliminată.

Acest comportament funcționează bine pentru majoritatea aplicațiilor, dar uneori se dorește să se urmărească obiectele doar atât timp cât sunt utilizate de altă aplicație. Din păcate această urmărire produce o legătură permanentă la memorie. Modulul `weakref` oferă instrumente pentru urmărirea obiectelor prin crearea unor legături slabe. Când un obiect nu mai este necesar va fi automat eliminat din tabela de legături slabe.

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                                # creaza o legatura
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                          # nu creaza legatura
>>> d['primary']                              # fetch the object if it is still alive
10
>>> del a                                    # elimina o referinta
>>> gc.collect()                             # activeaza "cosul de gunoi"
0
>>> d['primary']                              # intrarea a fost eliminata automat
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in -toplevel-
    d['primary']                              # entry was automatically removed
  File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

## 11.7 Instrumente de lucru cu listele

Multe structuri de date se contruiesc cu ajutorul listelor predefinite. Oricum, uneori se impune necesitatea unei implementări alternative cu performanțe diferite de cele obișnuite.

Modulul `array` oferă obiectul `array()` care se comporta ca o listă care cuprinde *numai* obiecte de același fel, dar care sunt stocate mai compact decât în cazul listei standard. În următorul exemplu se prezintă cum un *array*<sup>1</sup> de numere memorate ca numere binare fără semn reprezentate pe doi octeți (cod de tip "H") este contrapus utilizării unei liste standard de obiecte Python de tip `int`, reprezentate pe 16 octeți :

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

Modulul `collection` conține obiectul `deque()` care se aseamănă cu tipul listă, cu adăugări și scoateri rapide de elemente prin stânga, dar cu o accesare lentă a mijlocului. Aceste obiecte sunt convenabile pentru implementarea coziilor și a căutărilor pe nivel pentru arbori :

---

<sup>1</sup> Array = Lista cu valori de același tip care se pot referi individual

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

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

Biblioteca oferă, tot pentru alternativa la implementarea cu liste, și modulul `bisect` cu funcții pentru manipularea listelor sortate :

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

Modulul `heapq` conține funcții pentru implementarea de *heap*<sup>2</sup> utilizând listele standard. Valoarea intrării celei mai mici este ținută permanent la poziția zero. Acest lucru este util pentru aplicațiile în care se accesează repetitiv elementul cel mai mic, dar nu se dorește o sortare completă a listei:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearanjeaza lista in ordinea heap
>>> heappush(data, -5) # se adauga o noua intrare
>>> [heappop(data) for i in range(3)] # aduce cele mai mici trei intrari
[-5, 0, 1]
```

## 11.8 Aritmetica în virgulă flotantă zecimală

Modulul `decimal` oferă, pentru aritmetica în virgulă flotantă, tipul de data `Decimal`. Comparând-o cu clasa predefinită `float`, implementată pentru lucrul în virgulă flotantă binară, noua clasă este necesară în aplicațiile financiare și, în general, în aplicațiile care necesită : o reprezentare zecimală exactă, o precizie ridicată, o rotunjire impusă foarte bună, poziționarea zecimalelor semnificative. O altă întrebare ar fi pentru aplicațiile în care utilizatorul dorește confruntarea calculelor cu cele executate manual.

De exemplu, calculând taxa de 5% pentru o convorbire telefonică de 70 de cenți, folosind aritmetica în virgulă flotantă zecimală și cea binară se obțin rezultate diferite :

<sup>2</sup> Heap = porțiune de memorie rezervată pentru a fi utilizată de un program ca zonă de stocare temporară a unor structuri de date a căror mărime, sau existență nu se pot determina decât în timpul rulării programului.

```
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.7349999999999999
```

În exemplu, rezultatul funcției `Decimal` menține la sfârșit un zero pentru ca partea zecimală să fie compusă din patru cifre. Astfel se simulează calculul făcut cu mâna, când înmulțind doua numere, fiecare cu două zecimale se obține la rezultat partea zecimală din patru cifre. În acest fel se evită situația care apare când aritmetica în virgulă flotantă binară nu reprezintă precis cantitatea zecimală.

Pentru că returnează rezultatul exact clasa `Decimal` este folosită în calcule modulo și în teste de egalitate, domenii pentru care aritmetica în virgulă flotantă binară nu este aplicabilă :

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

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

Modulul `Decimal` poate executa calcule cu o precizie impusă de utilizator :

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





## Continuarea?

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

Ar trebui să citiți, sau cel puțin să răsfoiți *Python Library Reference*, document ce explică foarte detaliat fiecare funcție, tip de date, modul, etc. existent în Python. Distribuția standard de Python conține o cantitate imensă de cod scris atât în C, cât și în Python, există module UNIX ce pot fi folosite pentru a citi căsuțele de poștă 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 documentații bucăți de cod scrise de alți programatori și link-uri legate de Python.

Copii ale acestui site există pe serverele din întreaga lume (Europa, Australia, Japonia, etc). În funcție de locul în care vă aflați, pentru o viteză de acces mai mare, puteți 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 puteți folosi grupul `comp.lang.python`, sau puteți scrie la `Python-list@python.org`. În fiecare zi sunt puse în jur de 120 de întrebări care, de obicei, își găsesc răspunsul.

Înainte de a pune o întrebare, verificați lista întrebărilor puse frecvent (Frequently 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 alți programatori, vă pot ajuta de foarte multe ori în rezolvarea problemelor dumneavoastră.



# Editarea în linie de comandă și repetarea comenzilor anterioare

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

Facilitățile despre care vă vom vorbi sunt disponibile în versiunile UNIX și CygWin ale interpretorului.

Acest capitol nu va explica facilitățile 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 afișează promptul principal sau cel secundar.

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

Cele mai importante sunt:

- **C-A**(Control-A) mută cursorul la începutul liniei;
- **C-E**(Control-E) mută cursorul la sfârșitul liniei;
- **C-B** mută cursorul cu o poziție la stânga;
- **C-F** mută cursorul cu o poziție la dreapta;
- **BACKSPACE** șterge caracterul de la stânga cursorului;
- **C-D** șterge caracterul de la dreapta cursorului;
- **C-K** șterge tot ce urmează pe linie în dreapta cursorului;
- **C-Y** reface ultimul șir șters cu **C-K**;
- **C\_** anulează ultima comandă efectuată. Operația 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 instrucțiune aceasta este salvată la sfârșitul buffer-ului.

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

**C-M** înaintează la următoarea instrucțiune din buffer.

Orice linie din buffer poate fi modificată. Dacă o linie a fost modificată veți vedea un caracter '\*' în fața acesteia care indică că a fost modificată. O instrucțiune din buffer este executată dacă se apasă, evident, **ENTER**.

**C-R** pornește o căutare incrementală în buffer în sens invers.

**C-S** pornește o căutare în sens normal.

## A.3 Redefinirea tastelor funcționale

Tastele funcționale și alți parametri ai bibliotecii "Readline", pot fi redefiniți prin modificarea fișierului ".inputrc". Pentru a redefini funcția unei combinații de taste trebuie folosită sintaxa:

tastă:funcție sau "șir de caractere":funcție.

O opțiune poate fi setată folosind

set opțiune valoare nouă.

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 funcție pentru Tab este aceea de a insera Tab, nu funcția de completare Readline. Dacă se dorește însă utilizarea sa în funcția de completare se va pune :

```
Tab: complete
```

în ".inputrc". (Bine înțeles că acum va fi mai dificil de a scrie spațiile de aliniere)

Completarea automată a numelui unei variabile, sau a unui modul, este opțională. Pentru a o activa în modul de lucru interactiv adăugați în fișierul de startup următoarele :<sup>1</sup>

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

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

Un fișier startup mai cuprinzător poate arăta ca în exemplul următor. De notat că el își șterge numele pe care le crează în momentul în care nu-i mai sunt necesare. Acest lucru se petrece când fișierul de startup se execută în același domeniu de definiție ca și comenzile interactive, iar eliminarea numelor evită generarea unor efecte colaterale în mediul interactiv. Puteți găsi acest lucru convenabil pentru a păstra câteva module importate, cum ar fi `os`, și care se dovedesc necesare în majoritatea sesiunilor interpretorului.

<sup>1</sup> Python va executa conținutul unui fișier identificat prin variabila de mediu `PYTHONSTARTUP` în momentul pornirii interpretorului interactiv.

```

# La interpretorul interactiv Python se adauga functia de autocompletare
# si un fisier de tip jurnal pentru comenzi. Se cere Python 2.0+,
# readline. Autocompletarea este executata implicit prin tasta Esc (dar,
# tasta poate fi inlocuita - vezi documentatia readline).
#
# Puneti fisierul in ~/.pystartup, iar variabila de mediu sa-l
# indice, cum ar fi in bash "export PYTHONSTARTUP=/max/home/itamar/.pystartup".
#
# Retineti ca PYTHONSTARTUP nu expandeaza "~", deci va trebui
# indicata intraga 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 facilități reprezintă un pas enorm comparativ cu versiunile anterioare ale interpretorului; oricum au rămas câteva deziderate neîmplinite : ar fi frumos ca să fie sugerat un alineat corect la continuarea unei linii (analizorul sintactic știe dacă se va cere un alineat). Acest nou mecanism ar putea utiliza tabela de simboluri a interpretorului. Util ar fi și un mecanism care să verifice (sau să sugereze) potrivirea parantezelor, ghilimelelor, etc.



# Aritmetica în virgulă flotantă: rezultate și limitări

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

0.125

are valoarea  $1/10 + 2/100 + 5/1000$ , și în același fel fracția binară

0.001

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

Problema este ușor de înțeles întâi în baza 10. Să considerăm fracția  $1/3$ . O puteți aproxima cu o fracție în baza 10:

0.3

sau, mai precis,

0.33

sau, și mai precis,

0.333

și tot așa. Nu contează ce număr de digiți veți scrie, rezultatul nu va fi nicicând exact  $1/3$ , dar crește acuratețea aproximării lui  $1/3$ .

În același mod, nu contează câți digiți în baza 2 folosiți, valoarea zecimală 0.1 nu va fi reprezentată exact niciodată printr-o fracție. În baza 2,  $1/10$  este o fracție care se repetă la infinit

0.0001100110011001100110011001100110011001100110011001100110011...

Oprirea la un număr finit de biți dă o aproximare. Iată ce se afișează în situația următoare :

```
>>> 0.1
0.100000000000000001
```

Pe majoritatea mașinilor existente astăzi acest rezultat se afișează dacă introduceți 0.1 la prompterului Python-ului.

Se întâmplă așa pentru că numărul de biți utilizați de hardware să memoreze valoarea în virgulă flotantă variază de la o mașină la alta, iar Python listează doar o aproximație zecimală a valorii zecimale reale a aproximației binare memorată în mașină. Pe majoritatea mașinilor, dacă Python va tipări valoarea zecimală reală a aproximației binare memorate pentru 0.1, va afișa

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Prompterul Python utilizează funcția implicită `repr()` ca să obțină o versiune de șir pentru toate obiectele pe care le afișează. În cazul numerelor flotante `repr(float)` rotunjește valoarea zecimală la 17 digiți semnificativi, astfel

```
0.10000000000000001
```

`repr(float)` folosește 17 biți semnificativi pentru că sunt considerați suficienți pe majoritatea mașinilor, așa că relația `eval(repr(x)) == x` este exactă pentru toate numerele flotante finite  $x$ , dar rotunjirea la 16 biți nu este suficientă pentru a face relația adevărată.

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

Funcția internă Python `str()` generează numai 12 digiți semnificativi și poate că doriți să o folosiți modificată. Este des utilizată funcția `eval(str(x))` pentru a-l obține pe  $x$ , dar este mai placut să se afișeze :

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

E bine de știut că, în sens real, este o iluzie : în mașină valoarea nu este exact  $1/10$ , valoarea adevărată din mașină este rotunjită la *afișare*.

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

```
>>> 0.1
0.10000000000000001
```

apare tentația să utilizați funcția `round()` ca să trunchiați iarăși la un digit. Dar aceasta nu face nimic:

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

Problema este că valoarea memorată în virgulă 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 soluție.

Altă consecință a faptului că 0.1 nu este exact  $1/10$  este aceea că adăugând la 0.1 pe el însuși de 10 ori nu se obține 1.0:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
fs>>> sum
0.9999999999999999
```



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

Cum s-ar spune la sfârșit: "nu este ușor de răspuns". Totuși nu disperați din cauza virgulei flotante! Erorile operațiilor de flotantă din Python sunt moștenite de la virgula flotantă a hardware-ului și la majoritatea mașinilor sunt de un ordin mai mic de  $1/2^{53}$  pe operație. Acestea sunt mai mult decât acceptabile pentru cele mai multe aplicații, dar trebuie să rețineți că nu sunt datorate aritmeticii zecimale și că la fiecare operație în virgulă flotantă apare o eroare de rotunjire.

Până la apariția unor cazuri patologice, pentru cele mai multe situații de utilizare a aritmeticii în virgulă flotantă se obțin rezultatele așteptate, dacă pur și simplu se rotunjește rezultatul final afișat la numărul de zecimale dorit. Uzual funcția `str()` este suficientă, iar pentru un control mai riguros a se vedea discuția despre operatorul de format `%` : formatele `%g`, `%f` și `%e` sunt moduri flexibile și ușoare pentru afișarea rezultatelor.

## B.1 Erori de reprezentare

Secțiunea explică în detaliu exemplul "0.1" și arată cum se poate face o analiză exactă a cazurilor. În principal se urmărește familiarizarea cu reprezentarea binară în virgulă flotantă.

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

```
>>> 0.1
0.10000000000000001
```

Ce este asta ?  $1/10$  nu este reprezentat exact ca o fracție binară. Cele mai multe mașini utilizează astăzi (noiembrie 2000) aritmetica de virgulă flotantă IEEE-754 și cele mai multe platforme mapează flotantele Python în "dublă precizie" IEEE-754. Dublul 754 are 53 biți de precizie. Așa că la intrarea în calculator se chinuie să convertească  $0.1$  în cea mai apropiată fracție posibilă de forma  $J/2^N$ , unde  $J$  este un întreg de exact 53 de biți. Se rescrie

$$1 / 10 \approx J / (2^N)$$

ca

$$J \approx 2^N / 10$$

și reapelând  $J$  are exact 53 de biți (este  $\geq 2^{52}$  dar  $< 2^{53}$ ), și 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 biți. 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 decât jumătatea lui 10, cea mai bună aproximație se obține prin rotunjirea:

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

Astfel cea mai bună aproximație 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 câțul va fi un pic mai mic decât  $1/10$ . Dar în nici un caz nu poate fi *exact*  $1/10$ !

Deci, niciodată computerul nu va “arăta”  $1/10$ : ce va arăta este exact fracția dată mai jos, cea mai bună aproximație în 754 care poate fi dată:

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

Dacă se înmulțește fracția cu  $10^{**}30$ , vom vedea valoarea (truncheată) a celor mai semnificativi 30 de digiți zecimali ai săi:

```
>>> 7205759403792794L * 10L**30 / 2L**56
10000000000000000005551115123125L
```

însemnând că numărul exact memorat în calculator este aproximativ egal cu valoarea zecimală 0.100000000000000005551115123125. Rotunjind la 17 digiți semnificativi rezultă 0.10000000000000001 pe care Python îl afișează (va afișa pe orice platformă conformă 754 care va face tot posibilul să convertească intrare și ieșire prin bibliotecă C proprie — al dumneavoastră s-ar putea să nu reușească acest lucru!)

# Istoricul produsului Python și licențierea

## C.1 Istoricul produsului

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 contribuții ale altora.

În 1995 Guido își 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 și echipa de dezvoltare a Python-ului s-a mutat la BeOpen.com, unde a format echipa BeOpen PythonLabs. În octombrie 2000, colectivul PythonLabs s-a mutat la Digital Creations (acum Zope Corporation ; vezi <http://www.zope.com/>). În 2001 ia naștere Python Software Foundation (PSF, vezi <http://www.python.org/psf/>) o organizație non profit creată special pentru a deține drepturile de autor<sup>1</sup> referitoare la Python. Zope Corporation este unul din sponsorii PSF.

Toate lansările pe piață (versiunile) Python sunt Open Source (vezi <http://www.opensource.org/> pentru a găsi definiția Open Source ). Istoricește 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
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes

**Notă:** Compatibil GPL nu înseamnă că Python este distribuit sub GPL. Toate licențele Python, spre deosebire de

<sup>1</sup>În engleză : *Intellectual Property*

GPL, îți permit să distribui o versiune modificată fără a-ți face și modificările “open source”. Licențele compatibile GPL fac posibilă combinarea Python-ului cu alte software care au apărut sub GPL; celelalte neputând face acest lucru.

Mulțumim numeroșilor voluntari din exteriorul echipei, care lucrând sub coordonarea lui Guido, au contribuit la apariția acestor versiuni.

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

### **PSF LICENSE AGREEMENT FOR PYTHON 2.4.1**

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.4.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.4.1 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2004 Python Software Foundation; All Rights Reserved” are retained in Python 2.4.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.4.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.4.1.
4. PSF is making Python 2.4.1 available to Licensee on an “AS IS” basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.4.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.4.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.4.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.4.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

### **BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1**

1. This LICENSE AGREEMENT is between BeOpen.com (“BeOpen”), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (“Licensee”) accessing and otherwise using this software in source or binary form and its associated documentation (“the Software”).
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

#### **CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1**

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>.”
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

#### **CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2**

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

## **C.3 Licenses and Acknowledgements for Incorporated Software**

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

### **C.3.1 Mersenne Twister**

The `_random` module includes code based on a download from <http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.  
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`  
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:

1. Redistributions of source code must retain the above copyright  
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  
notice, this list of conditions and the following disclaimer in the  
documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote  
products derived from this software without specific prior written  
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR  
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR  
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.  
<http://www.math.keio.ac.jp/matsumoto/emt.html>  
email: [matumoto@math.keio.ac.jp](mailto:matumoto@math.keio.ac.jp)

## C.3.2 Sockets

The socket module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate  
source files from the WIDE Project, <http://www.wide.ad.jp/about/index.html>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND GAI\_ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR GAI\_ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON GAI\_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI\_ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:



```

/
      Copyright (c) 1996.
      The Regents of the University of California.
      All rights reserved.

Permission to use, copy, modify, and distribute this software for
any purpose without fee is hereby granted, provided that this en-
tire notice is included in all copies of any software which is or
includes a copy or modification of this software and in all
copies of the supporting documentation for such software.

This work was produced at the University of California, Lawrence
Livermore National Laboratory under contract no. W-7405-ENG-48
between the U.S. Department of Energy and The Regents of the
University of California for the operation of UC LLNL.

      DISCLAIMER

This software was prepared as an account of work sponsored by an
agency of the United States Government. Neither the United States
Government nor the University of California nor any of their em-
ployees, makes any warranty, express or implied, or assumes any
liability or responsibility for the accuracy, completeness, or
usefulness of any information, apparatus, product, or process
disclosed, or represents that its use would not infringe
privately-owned rights. Reference herein to any specific commer-
cial products, process, or service by trade name, trademark,
manufacturer, or otherwise, does not necessarily constitute or
imply its endorsement, recommendation, or favoring by the United
States Government or the University of California. The views and
opinions of authors expressed herein do not necessarily state or
reflect those of the United States Government or the University
of California, and shall not be used for advertising or product
endorsement purposes.
\

```

### C.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function.

License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this documentation and/or software.

### C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.6 Cookie management

The `Cookie` module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.  
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.8 Execution tracing

The trace module contains the following notice:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights...  
err... reserved and offered to the public under the terms of the  
Python 2.2 license.  
Author: Zooko O'Whielacronx  
<http://zooko.com/>  
<mailto:zooko@zooko.com>

Copyright 2000, Mojam Media, Inc., all rights reserved.  
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.  
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.  
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and  
its associated documentation for any purpose without fee is hereby  
granted, provided that the above copyright notice appears in all copies,  
and that both that copyright notice and this permission notice appear in  
supporting documentation, and that the name of neither Automatrix,  
Bioreason or Mojam Media be used in advertising or publicity pertaining to  
distribution of the software without specific, written prior permission.

### C.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse  
Cathedral City, California Republic, United States of America.  
All Rights Reserved  
Permission to use, copy, modify, and distribute this software and its  
documentation for any purpose and without fee is hereby granted,  
provided that the above copyright notice appear in all copies and that  
both that copyright notice and this permission notice appear in  
supporting documentation, and that the name of Lance Ellinghouse  
not be used in advertising or publicity pertaining to distribution  
of the software without specific, written prior permission.  
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO  
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND  
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE  
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES  
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN  
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT  
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:  
- Use binascii module to do the actual line-by-line conversion  
between ascii and binary. This results in a 1000-fold speedup. The C  
version is still 5 times faster, though.  
- Arguments more compliant with python standard

### C.3.10 XML Remote Procedure Calls

The `xmlrpc.lib` module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB

Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.