Python ro:Primii paşi

Contents

Introducere

Vom afla acum ce trebuie făcut pentru a rula tradiţionalul program ‘Hello World’ în Python. Astfel vom învăţa cum să scriem, salvăm şi rulăm programe Python.

Sunt doua căi de a folosi Python pentru a rula un program – folosind promptul interactiv al interpretorului sau folosind fişierul sursă. Vom afla cum se folosesc ambele metode.

Folosind promptul interpretorului

Porniţi interpretorul de la linia de comandă introducând python la prompt.
Pentru utilizatorii de Windows, puteţi rula interpretorul din linia de comandă dacă aveţi setată corect variabila PATH.
Dacă folosiţi IDLE (de la Integrated Developpement Linux Environment), dati clic pe Start→ Programs → Python 3.0 → IDLE (Python GUI).
Acum introduceţi print('Hello World') urmat de tasta Enter. Ar trebui să vedeţi ca rezultat cuvintele Hello World.

```
   $ python

   Python 3.0b2 (r30b2:65106, Jul 18 2008, 18:44:17) [MSC v.1500 32 bit
(Intel)] on win32

   Type "help", "copyright", "credits" or "license" for more information.

   >>> print('Hello World')

   Hello World

   >>>
```

Observaţi că Python va da rezultatul comenzii imediat! Ceea ce tocmai aţi introdus este o*declaraţie* Python singulară. Folosim print pentru (nesurprinzator) a tipări orice valoare pe care i-o furnizăm. Aici îi furnizăm textul Hello World şi acesta este tipărit rapid pe ecran.

Cum să părăsiţi interpretorul

Ca să părăsiţi prompt-ul, tastaţi Ctrl-D dacă folosiţi IDLE sau un shell Linux/BSD. În cazul consolei Windows (Command prompt), tastaţi Ctrl-Z urmat de tastaENTER.

Alegerea unui editor

Înainte de a trece la scrierea de programe Python În fişiere sursă avem nevoie de un editor pentru a crea aceste fişiere. Alegerea editorului este crucială. Trebuie ales la fel ca şi maşinile. Un editor bun vă va ajuta să scrieţi programe Python uşor, făcând timpul petrecut o călătorie confortabilă şi vă va ajuta să ajungeţi la destinaţie (să vă atingeţi obiectivul) întro maniera rapidă şi sigură.

O cerinţă de bază este **evidenţierea sintaxei** [1] în care diferitele componente ale sintaxei sunt colorate de aşa natură încât să poţi *vizualiza* programul şi rularea lui.
Dacă utilizaţi Windows, vă recomand să folosiţi IDLE. IDLE face syntax highlighting şi multe altele printre care faptul că vă permite să rulaţi programele tot în IDLE. O notă specială: **Nu folosiţi Notepad** – este o opţiune rea fiindcă nu face syntax highlighting şi nu suportă indentarea textului, ceea ce este foarte important în cazul nostru, aşa cum veţi vedea în continuare. Editoarele bune precum IDLE (şi VIM) vă vor ajuta automat să indentaţi textul.

Dacă utilizaţi Linux/FreeBSD, atunci aveţi o mulţime de opţiuni pentru editor. Dacă sunteţi chiar la începutul carierei de programator, poate o să preferaţi ‘geany’. Are interfaţă grafică cu utilizatorul şi butoane speciale pentru compilat şi rulat programele Python fără complicaţii.

Dacă sunteţi programator experimentat, atunci probabil că folosiţi deja Vim sau Emacs. Nu mai e nevoie să precizăm că acestea două sunt cele mai puternice editoare şi veţi avea nenumărate avantaje din folosirea lor la scrierea de programe Python. Eu personal folosesc Vim pentru majoritatea programelor. Dacă sunteţi programator începător, puteţi folosi Kate care este unul din favoritele mele. În cazul în care doriţi să alocaţi timpul necesar învăţării lucrului cu Vim sau Emacs, vă recomand să le învăţaţi pe amândouă, întrucât pe termen lung veţi culege foloase mult mai mari.
În această carte vom folosi **IDLE**, editorul nostru IDE cel mai recomandat. IDLE este instalat în mod implicit de către installerele Python pentru Windows şi Mac OS X. Este disponibil şi pentru Linux şi BSD în colecţiile (‘engl. repositories’) respective.
Vom explora folosirea mediului IDLE în capitolul următor. Pentru mai multe detalii, vă rog să vizitaţi documentaţia IDLE .
Dacă tot mai doriţi să vedeţi şi alte opţiuni pentru editor, recomand cuprinzătoarea listă de editoare pentru Python şi să optaţi. Puteţi alege şi un IDE (Integrated Development Environment)

pentru Python. A se vedea lista de medii integrate (IDE) care suportă Pythonpentru detalii suplimentare. Imediat ce veţi începe să scrieţi programe Python mari, IDE-urile pot fi cu adevărat foarte folositoare.
Repet, vă rog să alegeţi un editor adecvat – el poate face scrierea de programe Python mai distractivă şi uşoară.

Pentru utilizatorii de Vim

Există o introducere bună despre 'Cum să faci Vim un IDE puternic pentru Python'de John M Anderson.

Pentru utilizatorii de Emacs

Există o introducere bună despre 'Cum să faci Emacs un IDE puternic pentru Python' de Ryan McGuire.

Folosind un fişier sursă

Să ne întoarcem la programare. Există o tradiţie ca de câte ori înveţi un nou limbaj de programare, primul program pe care îl scrii să fie programul 'Hello World' – tot ce face el este să afişeze 'Hello World' când îl rulezi. După expimarea lui Simon Cozens [2], este 'incantaţia tradiţională către zeii programării ca sa te ajute să înveţi limbajul mai bine':) .
Porniţi editorul ales, introduceţi programul următor şi salvaţi-l sub numele helloworld.py
Dacă folosiţi IDLE, daţi clic pe File → New Window şi introduceţi programul de mai jos. Apoi clic pe File → Save.

```
#!/usr/bin/python
#Fisier: helloworld.py

print('Hello World')
```

Rulaţi programul deschizând un shell [3] şi introducând comanda python helloworld.py.
Daca folosiţi IDLE, deschideţi meniul Run → Run Module sau direct F5 de pe tastatură.
Rezultatul este afişat astfel:

```
$ python helloworld.py

Hello World
```

Daca aţi obţinut rezultatul afişat mai sus, felicitări! – aţi rulat cu succes primul program în Python.

În caz ca aţi obţinut un mesaj de eroare, vă rog, tastaţi programul anterior *exact* ca în imagine şi rulaţi programul din nou. De reţinut că Python este case-sensitive [4] aşadarprint nu este acelaşi lucru cu Print – observaţi p minuscul în primul exemplu şi P majuscul în al doilea exemplu. De asemenea, asiguraţi-vă că nu există spaţii sau TAB înaintea primului caracter din fiecare linie – vom vedea mai târziu de ce este atât de important.

CUM FUNCŢIONEAZĂ

Să considerăm primele două linii din program. Acestea sunt numite *comentarii* – orice s-ar afla la dreapta caracterului # devine comentariu şi este util în special pentru documentarea cititorului programului.

Python foloseşte comentarii numai pentru acest caz. Prima linie este numita linie *shebang*– de fiecare dată când începe cu #! urmată de locaţia unui program; asta spune sistemului nostru Linux/Unix că fişierul trebuie înteles prin acest interpretor atunci când este *executat*. O explicaţie mai detaliată va fi prezentată în capitolele următoare. De reţinut că puteţi rula oricând programul pe orice platformă specificând interpretorul în linia de comandă, ca în exemplul python helloworld.py .

> **Important**
>
> Folosiţi cu grijă comentarii în programe pentru a explica detalii importante ale unor instrucţiuni – Asta va ajuta cititorul să înţeleagă mai uşor ce ‘face’ programul. Acel cititor puteţi fi dumneavoastră, peste şase luni!

Comentariile sunt urmate de o *declaraţie* Python. În cazul nostru apelăm *funcţia* print care pur şi simplu tipăreşte pe ecran textul 'Hello World'. Vom învăţa despre funcţii într-un alt capitol; ce trebuie reţinut acum este că orice am fi pus în paranteze ar fi aparut pe ecran. În acest caz punem 'Hello World', ceea ce se poate numi *string* – fiţi fără grijă, vom explora mai târziu terminologia aceasta în detaliu.

PROGRAME PYTHON EXECUTABILE

Partea aceasta se aplică numai utilizatorilor de Linux/Unix, dar utilizatorii de Windows ar putea fi curioşi în legătură cu prima linie din program. Pentru început, va trebui să dăm fişierului permisiunea de a fi executabil folosind comanda chmod şi apoi să *rulăm*programul sursă.

```
$ chmod a+x helloworld.py

$ ./helloworld.py

Hello World
```

Comanda chmod este folosită aici pentru a schimba [5] *mod*-ul fişierului dându-i drept de execuţie pentru toţi [6] utilizatorii sistemului. Pe urmă executăm programul direct, specificând locaţia programului sursă. Folosim ./ pentru a indica localizarea programului executabil în directorul curent.

Pentru a face lucrurile şi mai distractive, puteţi redenumi fişierul cu numele helloworld şi îl puteţi rula cu ./helloworld şi tot va merge, folosind interpretorul de la locaţia specificată pe primul rând din fişier..

Ce e de făcut dacă nu ştim unde este localizat Python? Atunci puteţi folosi programul envspecific sistemelor Linux. Modificaţi primul rând astfel:

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

Programul env la rândul lui va căuta interpretorul Python care va rula programul.
Până acum am putut să executăm programele noastre doar dacă ştiam calea exactă. Dar dacă dorim să rulăm programul din orice director? Putem să facem asta dacă memorăm programul întrunul din directoarele listate în variabila de mediu PATH. Oridecâte ori rulaţi vreun program, sistemul caută acel program în directoarele listate în variabila PATH şi apoi rulează programul. Putem face programul nostru disponibil în orice director prin copierea programului întrunul din directoarele din PATH.

```
$ echo $PATH

/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin

$ cp helloworld.py /home/swaroop/bin/helloworld

$ helloworld

Hello World
```

Putem afişa conţinutul variabilei PATH folosind comanda echo şi prefixând numele variabilei cu caracterul $ pentru a-i transmite shell-ului că avem nevoie de valoarea acestei variabile. Observăm că /home/swaroop/bin este printre directoarele din PATH, unde*swaroop* este numele de utilizator [7] pe care eu îl folosesc în sistemul meu. Există unul similar pentru numele de utilizator al fiecaruia pe sistemul său. Ca alternativă, puteţi adăuga un director anume la variabila PATH – se face executândPATH=$PATH:/home/swaroop/mydir unde '/home/swaroop/mydir' este directorul pe care eu vreau sa-l adaug la variabila PATH.
Această metodă este foarte utilă dacă vreţi să scrieţi scripturi utile pe care vreţi să le rulaţi oricând din orice locaţie a sistemului. Seamănă cu a-ţi crea propriile comenzi, precum cdsau orice altă comandă pe care o executi în terminalul Linux sau DOS prompt.

Atenţie

Din punctul de vedere al Python-ului, program sau script sau software înseamnă acelaşi lucru!

Cum obţinem ajutor

Dacă aveţi nevoie repede de informaţii despre vreo funcţie sau declaraţie din Python, atunci puteţi apela la functionalitatea inclusă [8] help. Este foarte folositor, mai ales la promptul interpretorului. De exemplu rulaţi help(print) – se va afişa documentaţia de asistenţă pentru funcţia print folosită pentru afişarea pe ecran.

Notă

Tastaţi q pentru a ieşi din help.

Similar, puteţi obţine informaţii despre aproape orice din Python. Folosiţi help() pentru a afla mai multe despre însuşi help!

În cazul în care aveţi nevoie de ajutor în legătură cu operatori precum return, atunci va trebui să-i puneţi în ghilimele (ca în help('return')) pentru ca Python să înţeleagă fără confuzie ce încercaţi să faceţi.

Rezumat

Acum ar trebui să puteţi scrie, salva şi rula cu uşurinţă programe Python. Pentru ca aţi devenit utilizator de Python, să mai învăţăm câteva concepte din Python.

Python ro:Elemente

Simpla tipărire a textului 'Hello World' nu ajunge, aşa-i? Vreţi să faceţi mai mult de atât – vreţi să preluaţi ceva intrări, să le prelucraţi şi să obţineţi un rezultat. Putem face asta în Python folosind constante şi variabile.

Contents

Constante literale

O constanta literală este un număr precum 5, 1.23, 9.25e-3 sau un şir [1] precum 'Acesta este un şir' sau "E string!". Se numeşte literal fiindcă este folosit *literal* – îi folosim valoarea literalmente. Numărul 2 se reprezintă întotdeauna pe sine şi nimic altceva – este o*constantă* deoarece valoarea sa nu poate fi schimbată. De aici denumirea de constante literale.

Numere

Numerele în Python sunt de trei tipuri – *integer*, *float* şi *complex*.

- Un exemplu de integer (rom. întreg) este 2 care este un număr întreg.
- Exemple de *float* sau *floating point* [2] sunt 3.23 şi 52.3E-4. Notaţia E indică puterile lui 10. În acest caz, 52.3E-4 înseamnă $52.3 * 10^{-4}$.
- Exemple de numere complexe sunt (-5+4j) şi (2.3 - 4.6j)

Notă pentru programatorii experimentaţi

Nu există un tip separat 'long int'. Tipul implicit integer poate fi orice valoare mare.

Şiruri

Un şir (engl. string) este o *secvenţă* de *caractere*. Şirurile sunt în esenţă doar o succesiune de cuvinte. Cuvintele pot fi în limba engleză sau în orice altă limbă suportată de standardul Unicode, ceea ce înseamnă aproape orice limbă din lume.

Notă pentru programatorii experimentaţi

Nu există şiruri "ASCII" pure pentru că Unicode este un superset al ASCII. Dacă se impune în program un flux de octeţi codat ASCII, atunci folosiţi str.encode("ascii"). Pentru detalii, urmăriţi discuţia pe acest subiect de la StackOverflow.
Implicit, toate şirurile sunt în Unicode.

Aproape pot garanta că veţi folosi şiruri în aproape toate programele Python pe care le scrieţi, aşa că acordaţi atenţie părţii următoare despre cum se folosesc şirurile în Python.

GHILIMELE SIMPLE

Puteţi specifica şiruri folosind ghilimele simple [3] precum 'Citeaza-ma referitor la acest subiect'. Tot spaţiul alb precum SPACE şi TAB sunt păstrate ca atare [4].

GHILIMELE DUBLE

Şirurile în ghilimele duble [5] funcţioneaza la fel ca şi cele în ghilimele simple. De exemplu"Cum te cheamă?"

GHILIMELE TRIPLE

Puteţi specifica şiruri care se întind pe mai multe linii folosind ghilimele triple (engl. triple quotes) – (“”" sau ”’). Puteţi folosi liber ghilimele simple dau duble în interiorul ghilimelelor triple. Iată un exemplu:

```
'''Acesta este un şir multi-linie. Aceasta este prima linie.

Aceasta este a doua linie.

'Cum te numeşti?', l-a intrebat.

El a zis "Bond, James Bond".

'''
```

SECVENŢE DE EVADARE

Să presupunem că vrem să utilizăm un şir care conţine un apostrof (una din ghilimele simple) '. Cum o să specificăm şirul What's your name?. Nu-l putem specifica 'What's your name?' pentru că Python va confunda apostroful cu sfârşitul şirului. Într-un fel sau altul va trebui să specificăm că acel apostrof face parte din şir, nu este un delimitator. Pentru aceasta se foloseşte o *secvenţă de evadare* [6]. Specificăm apostroful \' – observaţi backslash-ul. Astfel putem specifica şirul 'What\'s your name?'.
Altă cale de a specifica acest şir este utilizarea ghilimelelor duble pentru delimitarea sirului. Problema apare şi la includerea unei ghilimele duble într-un şir delimitat cu ghilimele duble. Şi pentru evadarea backslash-ului trebuie tot backslash \\.
Dar dacă am vrea să specificăm un şir pe două rânduri? O soluţie este să folosim ghilimele triple cum am văzut mai devreme, dar putem să folosim o secvenţă de evadare pentru sfârşitul liniei – \n pentru a indica trecerea pe o linie nouă. Un exemplu ar fi Aceasta este prima linie\nAceasta este a doua linie. Alt exemplu util de secvenţă de evadare este pentru TAB – \t. Există mult mai multe, dar le-am prezentat aici pe cele mai folosite de noi.
Un lucru notabil este că într-un şir un backslash la sfârşitul liniei arată continuarea şirului pe linia următoare, fără să se adauge caracterul *newline*. De exemplu:

```
"Aceasta este prima propoziţie. \

Aceasta este a doua propoziţie".
```

este echivalent cu "Aceasta este prima propoziţie. Aceasta este a doua propoziţie".

ŞIRURI BRUTE

Dacă aveţi nevoie să specificaţi şiruri în care să nu fie procesate secvenţe de evadare trebuie să folosiţi şiruri *brute* [7] prefixând şirul cu r sau R. De exemplu r"Caracterul newline este indicat de \n".

ŞIRURILE SUNT IMUABILE

Asta înseamnă că odată create, nu mai pot fi modificate. Deşi pare un lucru rău, nu este. Vom vedea în diferitele programe prezentate de ce asta nu este o limitare.

CONCATENAREA LITERALILOR ŞIR

Dacă se alătura doi literali, ei sunt concatenaţi de Python automat. De exemplu 'What\'s ' 'your name?' este convertit automat în "What's your name?".

Notă pentru programatorii C/C++

Nu există în Python un tip de date separat char. Nu există nici o nevoie reală de aşa ceva, deci sunt sigur că n-o să-i duceţi dorul.

Notă pentru programatorii Perl/PHP

Reţineţi ca şirurile delimitate de gihilimele simple sau duble sunt la fel, nu diferă prin nimic.

Notă pentru utilizatorii de expresii regulare

Folosiţi întotdeauna şiruri brute cand aveţi de-a face cu expresii regulare, altfel o să fie nevoie de multe căutări în urmă pentru a găsi ce nu merge. De exemplu referinţele retroactive [8] pot fi utilizate ca '\\1' sau r'\1'.

METODA FORMAT

Uneori vrem să construim şiruri din alte informaţii. Aici este folositoare metoda format().

```
#!/usr/bin/python
# Fişier: str_format.py

vârstă = 25
nume = 'Swaroop'

print('{0} are {1} de ani.'.format(nume, vârstă))
print('De ce se joacă {0} cu python-ul ăla?'.format(nume))
```

Rezultat:

```
$ python str_format.py

Swaroop are 25 de ani.

De ce se joacă Swaroop cu python-ul ăla?
```

Cum funcţionează:

Un şir poate folosi anumite specificaţii şi apoi poate apela metoda *format* pentru a substitui acele specificaţii care corespund argumentelor metodei format.
Observaţi prima folosire, unde folosim {0} şi asta corespunde cu varibila nume care este primul argument al metodei format. Similar, a doua specificaţie este {1} corespunzatoare variabilei vârstă care este al doilea argument pentru metoda format.
Observaţi că puteam obţine acelaşi lucru prin concatenare: nume + ' are ' + str(vârstă) + ' de ani', dar uite ce urâtă şi predispusă la erori este această cale. În al doilea rând, conversia în şir este făcută automat de metoda format în locul unei conversii explicite. În al treilea rând, folosind metoda format putem schimba mesajul fără să avem de-a face cu variabilele şi reciproc.
Ce face Python în metoda format este că înlocuieşte valoarea fiecărui argument în locurile specificate. Pot exista şi specificări mai detaliate, cum ar fi:

```
>>> '{0:.3}'.format(1./3)  # zecimal (.) precizie de 3 zecimale pentru float
'0.333'
>>> '{0:_^11}'.format('hello')  # umple până la 11 caractere cu textul centrat şi bordat cu underscore (_)
'___hello___'
>>> '{nume} a scris {carte}.'.format(nume='Swaroop', carte='Un pic de Python')  # pe bază de cuvinte-cheie
'Swaroop a scris Un pic de Python.'
```

Detalii despre aceste specificaţii de formatare sunt date în PEP 3101 (PEP=Python Enhancement Proposal).

Variabile

Folosirea exclusiv a literalilor poate deveni rapid plictisitoare – avem nevoie să stocăm orice informaţie şi să o prelucrăm. Aici este locul *variabilelor*. Variabilele sunt exact ceea ce spune numele lor – valoarea lor poate fi modificată, va să zică se poate stoca orice întro variabilă. Variabilele sunt nişte mici zone din memoria calculatorului unde se stochează nişte informaţie. Spre deosebire de constante, e nevoie de a accesa această informaţie, din acest motiv variabilele au nume.

Nume de identificatori

Variabilele sunt exemple de identificatori. *Identificatorii* sunt nume date pentru a identifica*ceva*. Există câteva reguli care trebuie să fie urmate la stabilirea identificatorilor:

- Primul caracter al numelui trebui să fie o litera a alfabetului (majusculă ASCII, minusculă ASCII, caracter Unicode) sau underscore (‘_’).
- Restul numelui identificatorului poate include şi cifre (de la 0 la 9).
- Pentru numele de identificatori majusculele si minusculele sunt considerate diferite (engl. case-sensitive). De exemplu, myname şi myName **nu** desemnează aceeaşi variabilă. Observaţi minuscula n în primul caz şi majuscula N în al doilea.

- Exemple de nume *valide* de identificator
  sunt i, __chiar_aşa, nume_23, a1b2_c3 şiresumÃƒÆ'Ã†â€™Ãƒâ€
  Ã¢â‚¬â„¢Ã‚Æ'Ã¢â‚¬Å¡Ã‚Â©_count.
- Exemple de nume *invalide* de identificator sunt 2chestii, asta contine spaţii şi cal-breaz.

Tipuri de date

Variabilele pot lua valori de diferite tipuri numite **tipuri de date**. Tipurile de bază sunt numere şi şiruri, despre care am discutat deja. În ultimele capitole vom învăţa cum să creăm propriile noastre tipuri de date, folosind clase.

Obiecte

Reţineţi, Python consideră că tot ce se foloseşte în program este *obiect*, în sens generic. În loc de a spune *ceva-ul*, spunem *obiectul*.

Notă pentru utilizatorii de POO

Python este puternic orientat pe obiecte în sensul că toate sunt obiecte, inclusiv numerele, şirurile şi funcţiile.

Acum vom vedea cum se folosesc variabilele împreună cu literalii. Salvaţi următorul program şi rulaţi-l.

Cum se scriu programele Python

De acum încolo, procedura standard de a salva şi rula programele Python este astfel:

1. Deschideţi editorul preferat.
2. Introduceţi codul programului dat în exemplu.
3. Salvaţi-l întrun fişier cu numele menţionat în comentariu. Eu urmez convenţia de a salva toate programele Python în fişiere cu extensia .py.
4. Rulaţi-l folosind interpretorul cu comanda python program.py sau folosiţi IDLE pentru a rula programe. De asemenea puteţi folosi metoda executabilăcum am explicat mai devreme.

EXEMPLU: FOLOSIREA VARIABILELOR ŞI A LITERALILOR

```
# Fişier: var.py

i = 5
print(i)
i = i + 1
print(i)

s = '''Acesta este un şir multi-linie.
Aceasta este linia a doua.'''
print(s)
```

Rezultat:

```
$ python var.py

5

6

Acesta este un şir multi-linie.

Aceasta este linia a doua.
```

Cum funcţionează:

Iată cum lucrează programul: întâi, atribuim valoarea constantei literale 5 variabilei ifolosind operatorul de atribuire (=). Aceasta linie este o declaraţie deoarece susţine că trebuie făcut ceva, în acest caz, conectăm variabila i la valoarea 5. În continuare, tipărim valoarea lui i folosind declaraţia print care, previzibil, tipăreste valoarea variabilei pe ecran.
Apoi adăugăm 1 la valoarea stocată în i şi păstrăm noul rezultat. Tipărim valoarea variabilei şi obţinem ce am prevăzut, valoarea 6.
Similar, atribuim literalul şir variabilei s şi o tipărim.

Notă pentru programatorii în limbaje cu tipuri statice

Variabilele sunt folosite prin simpla atribuire a unei valori. Nu este necesară nici o declaraţie sau definiţie de tip de date.

Linii logice şi linii fizice

O linie fizică este ceea ce *vedeţi* când scrieţi programul. O linie logică este ceea ce *vede Python* ca o singură declaraţie. Python presupune implicit că fiecare *linie fizică*corespunde unei *linii logice*.

Un exemplu de linie logică este o declaraţie precum print('Hello World') – dacă aceasta este singură pe linie (cum se vede în editor), atunci ea corespunde şi unei linii fizice. Implicit, Python încurajează folosirea unei singure linii logice pe linia fizică (rând), ceea ce face codul mult mai lizibil.

Dacă vreţi să specificaţi mai mult de o linie logică pe linie fizică, va trebui să specificaţi explicit încheierea liniei logice cu (;). De exemplu,

```
i = 5

print(i)
```

este efectiv la fel ca

```
i = 5;

print(i);
```

şi acelaşi lucru poate fi scris

```
i = 5; print(i);
```

sau chiar

```
i = 5; print(i)
```

Totuşi, **recomand cu tărie** să rămâneţi la scrierea cel mult a unei **singure linii logice pe fiecare linie fizică**. Prin folosirea mai multor linii logice pe o linie fizică se obţine realmente cod mai lung. Ideea este să se evite semnul punct şi virgulă la maxim posibil pentru a obţine un cod cât mai lizibil. De fapt, eu n-am folosit *niciodată* şi nici n-am văzut punct şi virgulă într-un program Python.

Să dăm un exemplu de linie logică întinsă pe mai multe linii fizice, care se numeşte**reunire explicită a liniilor**.

```
s = 'Acesta este un şir \

care continuă pe a doua linie.'

print(s)
```

Se obţine rezultatul:

```
Acesta este un şir care continuă pe a doua linie.
```

Similar,

```
print\

(i)
```

este la fel ca

```
   print(i)
```

Există şi **reunire implicită a liniilor**, conform unei prezumţii care elimină nevoia de backslash. Este cazul în care linia logică foloseşte paranteze rotunde, paranteze drepte sau acolade. Le veţi vedea în acţiune când vom scrie programe folosind liste în capitolele finale.

Indentarea

Spaţiul alb este important în Python. De fapt, **spaţiul alb la începutul liniei este important**. Acesta se numeşte **indentare**. Spaţiul alb (spaţii şi taburi) de la începutul liniei logice este folosit pentru a determina nivelul de indentare al liniei logice, care la rândul lui este folosit pentru a determina gruparea declaraţiilor.

Asta înseamnă că declaraţiile care merg împreună **trebuie** să aibă aceeaşi indentare. Fiecare astfel de set de declaraţii se numeşte **bloc**. Vom vedea exemple despre importanţa blocurilor în capitolele următoare.

Un lucru demn de reţinut este că indentarea greşită poate produce erori. De exemplu:

```
i = 5
 print('Valoarea este ', i) # Eroare! Observaţi un spaţiu la începutul liniei.
print('Repet, valoarea este ', i)
```

Când rulaţi asta, obţineţi următoarea eroare:

```
    File "whitespace.py", line 4

      print('Valoarea este ', i) # Eroare! Observaţi un singur spaţiu la
începutul liniei.

      ^

  IndentationError: unexpected indent
```

Observaţi că există un spaţiu la începutul liniei a doua. Eroarea indicată de Python ne spune că sintaxa este greşită, adică programul nu a fost scris corespunzător. Asta înseamnă că *nu poţi începe în mod arbitrar noi blocuri de declaraţii* – cu excepţia blocului principal[9] implicit pe care l-aţi folosit tot timpul. Cazurile în care puteţi folosi un nou bloc de declaraţii vor fi detaliate în capitolele finale, cum ar fi capitolul despre controlul execuţiei.

Cum se indentează

Nu folosiţi un amestec de SPACE şi TAB fiindcă programele nu vor lucra corect pe toate platformele. Vă *recomand călduros* să folosiţi un *singur TAB* sau *patru spaţii* pentru fiecare nivel de indentare.

Alegeţi oricare din aceste stiluri de indentare. Şi mai important, alegeţi un stil şi folosiţi-l în mod **consistent** şi *exclusiv*.

Notă pentru programatorii în limbaje cu tipuri statice

Python va folosi mereu indentarea pentru blocuri şi niciodată acolade. Rulaţi from __future__ import braces pentru a afla mai multe detalii.

Rezumat

Acum că am trecut prin multe detalii esenţiale, putem continua cu lucruri mai interesante cum ar fi declaraţii pentru controlul execuţiei. Asiguraţi-vă că aţi înţeles ce aţi învatat în acest capitol.

Python ro:Operatori şi expresii

Contents

Introducere

Majoritatea declaraţiilor (linii logice) pe care le scrieţi conţin **expresii.** Un exemplu de expresie simplă este 2 + 3. O expresie poate fi descompusă în operatori şi operanzi.
Operatorii sunt functionalităţi care execută ceva şi pot fi reprezentaţi prin simboluri precum+ sau prin cuvinte cheie speciale. Operatorii au nevoie de nişte date asupra cărora să opereze, numite *operanzi*. În acest caz, 2 şi 3 sunt operanzii.

Operatori

Vom arunca o privire rapidă asupra operatorilor şi a folosirii lor:

Reţineţi că puteţi evalua expresiile date în exemple folosind interactiv interpretorul. De exemplu, pentru a testa expresia 2 + 3, folosind interactiv interpretorul Python:

```
>>> 2 + 3
5
>>> 3 * 5
15
```

```
>>>
```

OPERATOR	NUME	EXPLICAȚIE	EXEMPLE
+	Plus	adună două obiecte	3 + 5 fac 8 'a' + 'b' fac 'ab'.
-	Minus	fie face un număr să fie negativ fie dă diferența între două numere	-5.2 face negativ numărul 5.2 50 - 24 fac 26.
*	Inmulţire	dă produsul a două numere sau repetarea unui şir de numărul specificat de ori	2 * 3 fac 6 'la' * 3 dă 'lalala'.
**	Putere	dă x la puterea y	3 ** 4 dă 81 (adică 3 * 3 * 3 * 3)
/	Împărţire	împarte x la y	4 / 3 dă 1.3333333333333333.
//	Împărţire întreagă	dă partea întreagă a câtului	4 // 3 fac 1.
%	Modulo	dă restul împărţirii	8% 3 fac 2 -25.5% 2.25 fac 1.5.
<<	Translaţie la stânga	Translateaza biţii unui număr la stânga cu numărul specificat de biţi. (Orice număr este reprezentat în memorie sub forma de biţi – cifre binare 0 şi 1)	2 << 2 da 8 2 este reprezentat prin 10 în biţi. Prin translaţie la stânga cu doi biţi se obţine1000 ceea ce reprezintă numărul 8.
>>	Translaţie la dreapta	Translateaza biţii numărului la dreapta cu numărul specificat de biţi.	11 >> 1 dă 5 11 este reprezentat în biţi prin 1011care translatat la dreapta cu un bit dă101 ceea ce reprezintă numărul 5.
&	AND	ŞI binar între numere	5 & 3 da 1.

|	OR	SAU binar între numere	5 | 3 dă 7
^	XOR	SAU exclusiv binar între numere	5 ^ 3 fac 6
~	Complement binar	complementul lui x este -(x+1)	~5 dă -6.
<	Mai mic (decât)	Valoarea de adevăr a propoziţiei x este mai mic decât y. Toţi operatorii de comparaţie iau valorile logice True sau False. Observaţi că aceste nume încep cu majusculă.	5 < 3 dă False 3 < 5 dă True. Comparaţiile pot fi înlănţuite arbitrar: 3 < 5 < 7 dă True.
>	Mai mare (decât)	Valoarea de adevăr a propoziţiei x este mai mare decât y.	5 > 3 dă True. Dacă ambii operanzi sunt numere, aceştia sunt convertiţi întâi la un tip comun. În caz contrar operaţia ar avea mereu valoarea False.
<=	Mai mic sau egal (cu)	Valoarea de adevăr a propoziţiei x este mai mic sau cel mult egal cu y.	x = 3; y = 6; x <= y dă True.
>=	Mai mare sau egal (cu)	Valoarea de adevăr a propoziţiei x este mai mare sau cel puţin egal cu y.	x = 4; y = 3; x >= 3 da True.
==	Egal (cu)	Verifică dacă două numere sunt egale	x = 2; y = 2; x == y dă True. x = 'str'; y = 'stR'; x == y dă False. x = 'str'; y = 'str'; x == y dă True.
!=	Diferit (de)	Verifică dacă două numere sunt diferite	x = 2; y = 3; x!= y dă True.
not	NU logic	dacă x este True, dăFalse. Dacă x esteFalse, dă True.	x = True; not x dă False.
and	ŞI logic	x and y dă False dacă x este False, altfel dă valoarea lui y	x = False; y = True; x and y dă Falseîntrucât x este False. În acest caz, Python nu va evalua pe y

			fiindcă ştie că partea stângă a expresiei ‘and’ esteFalse ceea ce dă întregii expresii valoarea False indiferent de celelalte valori. Acest fapt se numeşte evaluare în circuit scurt.
or	SAU logic	dacă x este True, dăTrue, altfel dă valoarea lui y	x = True; y = False; x or y dă True. Şi aici se aplică evaluarea în circuit scurt.

Operatori şi folosirea lor

PRESCURTARE PENTRU OPERAŢII MATEMATICE ŞI ATRIBUIRI

Este uzual să faci o prelucrare matematică a unei variabile şi să păstrezi apoi rezultatul tot în ea; de aceea există o prescurtare pentru acest fel de expresii:

În loc de:

```
a = 2; a = a * 3
```

puteţi scrie:

```
a = 2; a *= 3
```

Observaţi că var = var operaţie expresie devine var operaţie= expresie.

Ordinea de evaluare

Daca aveţi o expresie precum 2 + 3 * 4, se va evalua întâi operaţia de adunare sau cea de înmulţire? Matematica de liceu ne învaţă că multiplicarea ar trebui făcută întâi. Asta înseamnă că operatorul de înmulţire are precedenţă mai mare decât operatorul de adunare.
Tabelul următor dă precedenţa operatorilor în Python, de la cea mai mică precedenţă (cea mai slabă legătură) până la cea mai mare precedenţă (cea mai strânsă legătură). Asta înseamnă că într-o expresie dată, Python va evalua întâi operatorii şi expresiile cele mai de jos în tabel înaintea celor mai de sus.

Următorul tabel, extras din manualul de referinţe Python, este dat de dragul completitudinii. Este de departe mai bine să folosim paranteze pentru a grupa operatorii şi operanzii în mod adecvat pentru a specifica precedenţa. Astfel programul devine mai lizibil. Pentru detalii vă rog să urmăriţi mai jos Schimbarea ordinii de evaluare.

OPERATOR	DESCRIERE
lambda	Expresie lambda

or	SAU logic
and	ŞI logic
not x	NU logic
in, not in	Teste de apartenenţă
is, is not	Teste de identitate
<, <=, >, >=,!=, ==	Comparaţii
|	SAU binar
^	SAU-exclusiv binar
&	ŞI binar
<<, >>	Translaţii
+, -	Adunare şi scădere
*, /, //, %	Înmulţire, împărţire, împărţire întreagă, modulo
+x, -x	Pozitiv, negativ
~x	NU binar
**	Exponenţiere

x.atribut	Referinţă la atribut
x[index]	Referinţă la element
x[index1:index2]	Feliere
f(argumente …)	Apel la funcţie
(expresii, …)	Legătura sau afişarea unui cuplu
[expresii, ...]	Afişarea unei liste
{cheie:date, …}	Afişarea unui dicţionar

Precedenţa operatorilor

Operatorii pe care nu i-am întâlnit până acum vor fi descrişi în capitolele viitoare.

Operatorii cu *aceeaşi precedenţă* sunt listaţi în acelaşi rând în tabelul anterior. De exemplu, + şi - au aceeaşi precedenţă.

Schimbarea ordinii de evaluare

Pentru a face expresiile mai lizibile, putem folosi paranteze. De exemplu, 2 + (3 * 4) este în mod clar mai uşor de înţeles decât 2 + 3 * 4 care necesită cunoaşterea precedenţei operatorilor. Ca şi orice altceva, parantezele trebuie folosite cu discernământ (nu exageraţi) şi fără redundanţă (ca în 2 + (3 + 4)).

Există un avantaj suplimentar în folosirea parantezelor – ne ajută să schimbăm ordinea de evaluare. De exemplu, dacă vreţi să fie evaluată adunarea înaintea înmulţirii întro expresie, trebuie să o scrieţi (2 + 3) * 4.

Asociativitatea

Operatorii sunt de obicei asociativi de la stânga la dreapta, adică operatorii cu aceeaşi precedenţă sunt evaluaţi de la stânga la dreapta. De exemplu, expresia 2 + 3 + 4 este evaluata ca (2 + 3) + 4. Câţiva operatori, precum atribuirea sunt asociativi de la dreapta la stânga astfel espresia a = b = c este evaluată caa = (b = c).

Expresii

Exemplu:

```
#!/usr/bin/python
# Fişier: expression.py

lungime = 5
lăţime = 2

aria = lungime * lăţime
print('Aria este', aria)
print('Perimetrul este', 2 * (lungime + lăţime))
```

Rezultat:

```
$ python expression.py

Aria este 10

Perimetrul este 14
```

Cum funcţionează:

Lungimea şi lăţimea dreptunghiului sunt stocate în variabile cu numele respective. Le folosim pentru a calcula aria şi perimetrul dreptunghiului cu ajutorul expresiilor. Stocăm rezultatul expresiei lungime * lăţime în variabila aria şi o afişăm folosind funcţia print. În al doilea caz, folosim direct valoarea expresiei 2 * (lungime + lăţime) în funcţia print.
De asemenea, observaţi cum Python 'cosmetizează' tipărirea rezultatului. Deşi noi n-am specificat un spaţiu între 'Aria este' şi variabila aria, Python o face pentru noi ca să obţinem o prezentare mai clară şi astfel programul este mult mai lizibil (fiindcă nu mai trebuie să ne îngrijim de spaţierea şirurilor folosite pentru afişare). Acesta este un exemplu despre cum face Python viaţa programatorului mai uşoară.

Rezumat

Am învăţat ce sunt operatorii, operanzii şi expresiile – acestea sunt componentele de bază ale oricarui program. În continuare vom vedea cum se folosesc în declaraţii.

Python ro:Controlul execuţiei

Contents

Introducere

În programele pe care le-am văzut până acum erau o serie de declaraţii şi Python le executa credincios în aceeaşi ordine. Dar dacă am fi vrut să schimbam fluxul sau modul lui de lucru? De exemplu, vreţi ca programul să ia nişte decizii şi să facă procesări diferite în diferite situaţii, precum a tipări 'Bună ziua' sau 'Bună seara' în funcţie de ora la care se execută programul?

Cum poate aţi ghicit, asta se poate face cu declaraţii de control al execuţiei. Există trei declaraţii de control al execuţiei în Python – if, for şi while.

Declaraţia if

Declaraţia if este folosită pentru a testa o condiţie şi, dacă aceasta este adevărată, să ruleze un bloc de declaraţii (numit 'blocul if'), iar în caz contrar să ruleze alt bloc de declaraţii (blocul 'else'). Clauza 'else' este optională.

Exemplu:

```
#!/usr/bin/python
# Fişier: if.py

număr = 23
ghici = int(input('Introduceţi un întreg: '))

if ghici == număr:
      print('Felicitări, aţi ghicit,')  # Noul bloc începe aici
      print('(dar nu câştigaţi niciun premiu!)')  # Noul bloc se încheie aici

elif ghici < număr:
      print('Nu, e un pic mai mare.')  # Alt bloc
      # Poti face ce vrei într-un bloc ...
else:
      print('Nu, e un pic mai mic.')
      # Ca să ajungeţi aici e sigur ca ghici > număr

print('Gata')
# Aceasta ultimă declaraţie este executată întotdeauna, după declaraţia if
```

Rezultat:

```
$ python if.py

Introduceţi un întreg: 50

Nu, e un pic mai mic.

Gata



$ python if.py
```

```
Introduceţi un întreg: 22

Nu, e un pic mai mare.

Gata



$ python if.py

Introduceţi un întreg: 23

Felicitări, aţi ghicit,

dar nu câştigaţi niciun premiu!

Gata
```

Cum funcţionează:

În acest program preluăm de la utilizator încercări de a ghici numărul şi verificăm dacă este numărul memorat. Setăm variabila număr la ce valoare vrem, să zicem 23. Apoi preluăm numărul încercat de utilizator folosind funcţia input(). Funcţiile sunt nişte porţiuni de program reutilizabile. Vom afla mai multe despre ele în capitolul următor.
Furnizăm un şir funcţiei implicite input() care îl tipăreşte pe ecran şi aşteaptă introducerea de informaţie de la utilizator. Îndată ce introducem ceva (ENTER – rom. a intra/introduce) şi apăsăm tasta ENTER, funcţia input() dă ca rezultat ceea ce am introdus, sub formă de şir. Convertim acest şir întrun întreg folosind declaraţia int şi stocăm valoarea în variabilaghici. De fapt int este o clasă, dar ce trebuie să ştiţi în acest moment este că îl folosiţi pentru a converti un şir într-un întreg (presupunând că şirul conţine un întreg valid în text).
În continuare comparăm alegerea utilizatorului cu numărul stabilit de noi. Dacă acestea sunt egale, tipărim un mesaj de succes. Observaţi că folosim nivele de indentare pentru a-i spune Pythonului cărui bloc aparţine fiecare declaraţie. Iată de ce este indentarea atat de importantă în Python. Sper că v-aţi ataşat de regula indentării consistente. Este aşa?

Observaţi cum declaraţia if conţine semnul două puncte la sfârşit – aşa îi spunem Pythonului că urmează un bloc de declaraţii.
Mai departe, testăm dacă numărul furnizat de utilizator este mai mic decât numărul şi, dacă este aşa, informăm utilizatorul că trebuie să ţintească mai sus de atât. Ce am folosit aici este clauza elif care de fapt combină două declaraţii if else-if else într-o singură declaraţie if-elif-else. Asta face programul mai uşor şi reduce numărul de indentări necesar.

Şi clauzele elif şi else trebuie să aibă la sfârşitul liniei logice semnul două puncte după care poate urma blocul lor de declaraţii (cu indentarea adecvată, desigur).
Puteţi pune o altă declaraţie if în interiorul blocului ‘if’ al declaraţiei if s.a.m.d. – în acest caz declaraţiile if se numesc *imbricate* (engl. nested).
Clauzele elif şi else sunt opţionale. O declaraţie if minimală este:

```
if True:
    print('Da, e adevarat.')
```

După ce Python a terminat execuţia întregii declaraţii if inclusiv clauzele elif şi else, el trece la următoarea declaraţie din blocul care conţine declaraţia if. În acest caz este vorba de blocul main (rom. principal), unde începe întotdeauna execuţia programului, iar instrucţiunea următoare este declaraţia print('Gata'). După aceasta, Python vede sfârşitul programului şi încheie.
Deşi acesta este un program foarte simplu, am indicat o mulţime de lucruri care trebuie observate. Toate acestea sunt destul de directe (şi simple pentru cei care au cunoştinţe de C/C++) şi iniţial necesită să deveniţi constienţi de ele, dar apoi vor deveni uzuale şi vă vor părea ‘naturale’.

Notă pentru programatorii în C/C++

Nu există declaraţia switch în Python. Puteţi utiliza declaraţia if..elif..else pentru a face acelaşi lucru (şi în unele cazuri, puteţi folosi o structură de date pentru a rezolva repede).

Declaraţia while

Declaraţia while ne permite să executăm repetat un bloc de declaraţii atât timp cât o condiţie rămâne adevărată. O declaraţie while este un exemplu de instrucţiune de ciclare. Poate avea şi clauza else.
Exemplu:

```
#!/usr/bin/python
# Fişier: while.py

număr = 23
ciclu = True

while ciclu:
    ghici = int(input('Introduceţi un întreg: '))

    if ghici == număr:
        print('Felicitări, aţi ghicit!')
        ciclu = False  # asta face ciclul să se întrerupă
    elif ghici < număr:
        print('Nu, este puţin mai mare.')
    else:
        print('Nu, este puţin mai mic..')
else:
    print('Bucla s-a încheiat.')
    # Aici puteţi face ce prelucrări vreţi

print('Gata')
```

Rezultat:

```
$ python while.py

Introduceţi un întreg: 50

Nu, este puţin mai mic.

Introduceţi un întreg: 22

Nu, este puţin mai mare

Introduceţi un întreg: 23

Felicitări, aţi ghicit.

Bucla s-a încheiat.

Gata
```

Cum funcţionează:

În acest program jucăm tot jocul cu ghicirea numărului, dar avantajul este ca utilizatorul poate continua încercările până când ghiceşte – nu trebuie să ruleze programul de fiecare dată, cum am facut în programul precedent. Ceea ce este chiar o demostraţie de declaraţie while.
Deplasăm declaraţiile input şi if în interiorul buclei while şi iniţializăm
variabila ciclu cu Trueînaintea buclei. La început testăm dacă variabila ciclu este True şi apoi continuăm cu executarea blocului *while*. După ce blocul a fost executat, condiţia este evaluată din nou şi, în acest caz, condiţia este variabila ciclu. Dacă este True, executăm blocul *while* din nou, altfel verificăm dacă există o clauză *else* ca s-o executăm.
Blocul else este executat atunci cand condiţia de ciclare devine False – asta poate fi chiar şi prima dată când se testează condiţia. Dacă exista un bloc else la bucla while, ea va fi întotdeauna executată, dacă nu se iese forţat din buclă cu o declaraţie break.
Valorile True şi False sunt numite booleene şi pot fi considerate a fi echivalente cu valorile1 şi respectiv 0.

Notă pentru programatorii în C/C++

Reţineţi că poate exista o clauză else la bucla while.

Bucla for

Declaraţia for..in este o declaraţie de ciclare care *iterează* elementele unei secvenţe de obiecte. Vom afla mai multe despre secvenţe în capitolele următoare. Ce trebuie ştiut acum este că o

secvenţă este pur şi simplu o colecţie ordonată de elemente.

Exemplu:

```
#!/usr/bin/python
# Fişier: for.py

for i in range(1, 5):
    print(i)
else:
    print('Bucla s-a terminat')
```

Rezultat:

```
$ python for.py

1

2

3

4

Bucla s-a terminat
```

Cum funcţionează:

În acest program, tipărim o *secvenţă* de numere. Generăm secvenţa cu ajutorul funcţiei predefinite range.

Noi dăm funcţiei range două numere şi ea ne dă secvenţa de numere începând cu primul număr şi până la cel de-al doilea. De exemplu, range(1,5) înseamnă secvenţa [1, 2, 3, 4]. Implicit, range are pasul 1. Dacă îi dăm şi un al treilea număr, range acela devine pasul secvenţei. De exemplu range(1,5,2) dă [1,3]. Reţineţi că gama de numere (engl. range) se extinde *până la* al doilea număr, dar *nu'* îl şi include.

Aşadar bucla for iterează peste acesta gamă – for i in range(1,5) este echivalent cu for i in [1, 2, 3, 4] ceea ce este ca şi cum s-ar atribui fiecare obiect din secvenţă lui i, pe rând, şi executarea blocului de declaraţii pentru fiecare valoare a lui i. În acest caz, nu facem altceva decât să tipărim valoarea obiectului.

Amintiţi-vă că clauza else este opţională. Când este inclusă, este executată întotdeauna o dată, după încheierea buclei for, cu excepţia cazului în care se întâlneşte o declaraţiebreak.

De reţinut că bucla for..in funcţionează pentru orice secvenţă. În acest caz avem doar o listă de numere, generată cu funcţia predefinită range, dar în general, putem folosi orice fel de secvenţă de orice fel de obiecte.

Notă pentru programatorii în C/C++/Java/C#

În Python bucla for este radical diferită de bucla for din C/C++. Programatorii C# vor reţine că bucla for din Python este similară cu bucla foreach din C#. Programatorii Java să observe că acelaşi lucru este similar cu for (int i: IntArray) în Java 1.5.
În C/C++, dacă vrei să scrii for (int i = 0; i < 5; i++), atunci în Python scrii doar for i in range(0,5). Aşa cum vedeţi, în Python bucla for este mai simplă, mai expresivă şi mai puţin predispusă la erori.

Declaraţia break

Declaraţia break este folosită pentru a *întrerupe* (engl. break) executarea unei declaraţii de ciclare, chiar şi dacă condiţia testată nu a devenit încă False sau secvenţa nu a fost parcursă complet.
O notă importantă este că dacă se *întrerupe* o bucla for sau while, nici clauza else **nu va fi executată**.
Exemplu:

```
#!/usr/bin/python
# Fişier: break.py

while True:
      s = (input('Introduceţi ceva:'))
      if s == 'quit':
            break
      print('Lungimea şirului introdus este', len(s))
print('Gata')
```

Rezultat:

```
$ python break.py

Introduceţi ceva: Programarea e mişto

Lungimea şirului introdus este 15

Introduceţi ceva: Când treaba e facută

Lungimea şirului introdus este 20

Introduceţi ceva: Dacă vrei să te şi distrezi:

Lungimea şirului introdus este 27

Introduceţi ceva:       foloseşte Python!

Lungimea şirului introdus este 17

Introduceţi ceva: quit

Gata
```

Cum funcţionează:

În acest program, preluăm datele de intrare în mod repetat de la utilizator şi tipărim lungimea fiecărui şir introdus. Prevedem şi o condiţie specială pentru oprirea programului, prin căutarea cuvântului 'quit'. Oprim programul prin *întreruperea* buclei şi ajungerea la sfârşitul blocului de declaraţii.
Lungimea şirului de intrare poate fi găsită folosind funcţia predefinită len.
Reţineţi că declaraţia break poate fi folosită şi cu declaraţia for.

POEZIA LUI SWAROOP DESPRE PYTHON

Ce am folosit aici drept intrare (de la utilizator) este un mini poem scris de mine, numit**Swaroop's Poetic Python** (în limba engleză):

```
Programming is fun

When the work is done

if you wanna make your work also fun:

    use Python!
```

Declaraţia continue

Declaraţia continue se foloseşte pentru a spune lui Python să treacă la următoarea iteraţie fără să execute instrucţiunile rămase din blocul declaraţiei de ciclare.
Exemplu:

```
#!/usr/bin/python
# Fişier: continue.py

while True:
    s = input('Introduceţi ceva: ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Prea puţin')
        continue
    print('Şirul introdus are lungime suficientă')
    # Faceţi alte procesări aici...
```

Rezultat:

```
$ python test.py

Introduceţi ceva: a

Prea puţin
```

```
Introduceţi ceva: 12

Prea puţin

Introduceţi ceva: abc

Şirul introdus are lungime suficientă

Introduceţi ceva: quit
```

Cum funcţionează:

În acest program acceptăm date de la utilizator, dar le procesăm doar dacă au cel puţin 3 caractere lungime. Aşadar, folosim funcţia len pentru a obţine lungimea şi, dacă aceasta este mai mică decât 3, sărim peste ce a mai rămas din iteraţia curentă folosind declaraţiacontinue. În caz contrar, restul declaraţiilor din buclă sunt executate şi putem să facem orice fel de procesare în zona unde este acel comentariu.
Retineţi că declaraţia continue funcţionează şi cu bucla for.

Rezumat

Am văzut cum se folosesc cele trei instrucţiuni de control al execuţiei – if, while şi forîmpreună cu asociatele lor, declaraţiile break şi continue. Acestea sunt unele dintre cele mai utilizate părţi din Python şi de aceea este esenţial să te obişnuieşti cu ele.
În continuare vom învăţa să construim şi să folosim funcţii.

Python ro:Funcţii

Contents

Introducere

Funcţiile sunt porţiuni de program reutilizabile. Ele vă permit să daţi nume unui bloc de declaraţii şi puteţi rula acel bloc de declaraţii în program de câte ori vreţi. Asta se numeşte*apel* al funcţiei. Noi am folosit deja multe funcţii predefinite precum len şi range.
Conceptul de funcţie este probabil *cel* mai important bloc constructiv al oricărui program nonbanal (în orice limbaj de programare), deci vom explora diverse aspecte ale funcţiilor în acest capitol.

Funcţiile sunt **def**inite folosind cuvântul cheie def. Acesta este urmat de un nume*identificator* pentru funcţie urmat de o pereche de paranteze care pot include nişte nume de variabile. În continuare este plasat blocul de declaraţii care compun funcţia. Un exemplu va arăta cât este de simplu:
Exemplu:

```
#!/usr/bin/python
# Fişier: function1.py

def sayHello():
      print('Hello World!')  # blocul funcţiei
# Sfârşitul funcţiei

sayHello()  # apel la funcţia sayHello()
sayHello()  # din nou apel la funcţia sayHello()
```

Rezultat:

```
   $ python function1.py

   Hello World!

   Hello World!
```

Cum funcţionează:

Definim o funcţie numită sayHello folosind sintaxa explicată mai sus. Aceasta funcţie nu primeşte parametri şi deci nu sunt declarate variabile în paranteze. Parametrii pentru funcţie sunt doar nişte modalităţi de a-i transmite funcţiei diferite valori sau/şi de a extrage valorile corespunzătoare. Observaţi că putem apela aceeaşi funcţie de două ori, ceea ce înseamnă că nu mai trebuie să scriem aceeaşi porţiune de cod din nou.

Parametrii funcţiilor
O funcţie poate accepta parametri, care sunt valori furnizate funcţiei pentru ca aceasta să poată *face* ceva cu aceste valori. Aceşti parametri sunt ca variabilele numai că valorile acestor

variabile sunt definite în momentul apelului funcţiei şi deja le sunt atribuite valori la momentul executării blocului funcţiei.

Parametrii sunt specificaţi într-o pereche de paranteze în definiţia funcţiei, separate prin virgule. Când apelăm funcţia, furnizăm aceste valori într-un fel sau altul. Observaţi terminologia folosită – numele date în funcţie se numesc *parametri* în timp ce valorile pe care le furnizăm în apelul funcţiei se numesc *argumente*.

Exemplu:

```
#!/usr/bin/python
# Fişier: func_param.py

def printMax(a, b):
    if a > b:
        print(a, 'este maximum')
    elif a == b:
        print(a, 'este egal cu', b)
    else:
        print(b, 'este maximum')

printMax(3, 4)  # argumente date prin literali

x = 5
y = 7

printMax(x, y)  # argumente date prin variabile
```

Rezultat:

```
$ python func_param.py

4 este maximum

7 este maximum
```

Cum funcţionează:

Aici definim o funcţie numită printMax care primeşte doi parametri numiţi a şi b. Găsim cel mai mare număr dintre ele folosind o simpla declaraţie if..else şi tipărim pe ecran cel mai mare număr. În primul apel al funcţiei printMax, furnizăm argumentele în forma literală. În al doilea apel dăm funcţiei valorile parametrilor prin intermediul variabilelor. printMax(x, y) face ca valoarea variabilei x să fie atribuită parametrului a şi valoarea variabilei y să fie atribuită parametrului b. Funcţia printMax lucrează la fel în ambele cazuri.

Variabile locale

Când se declară variabile în interiorul definiţiei funcţiei, acestea nu au nici un fel de legătură cu alte variabile din afara definiţiei funcţiei, nici chiar dacă ar avea acelaşi nume, de aceea se numesc variabile *locale* funcţiei. Acesta este *domeniul* variabilei. Toate variabilele au ca domeniu blocul în care sunt declarate, începând cu punctul în care a fost definit numele ei.

Exemplu:

```
#!/usr/bin/python
# Fişier: func_local.py

x = 50

def func(x):
      print('x este', x)
      x = 2
      print('Am schimbat x local în ', x)

func(x)
print('x este tot ', x)
```

Rezultat:

```
   $ python func_local.py

   x este 50

   Am schimbat x local în 2

   x este tot 50
```

Cum funcţionează:

În funcţie, prima dată când folosim *valoarea* numelui x, Python foloseşte valoarea parametrului declarat în funcţie.
În continuare atribuim valoarea 2 lui x. Numele x este local funcţiei noastre. Prin urmare, când schimbăm valoarea lui x în funcţie, x definit în blocul principal rămâne neafectat.
În ultimul apel al funcţiei print, afişăm valoarea lui x din blocul principal ca să confirmăm că a rămas neafectată.

Folosirea declaraţiei global

Dacă vreţi să atribuiţi o valoare unui nume definit la nivelul cel mai înalt al programului (adică nu în interiorul domeniului funcţiei sau clasei), va trebui să-i spuneţi lui Python că acel nume nu este local ci *global*. Obţinem asta folosind declaraţia global. Este imposibil ca în interiorul unei funcţii să atribui o valoare unei variabile definite în afara funcţiei fără declaraţia global.

Puteţi folosi valorile definite în afara funcţiilor (presupunând că nu există o variabilă cu acelaşi nume definită în blocul funcţiei). Totuşi, acest fapt nu este încurajat şi trebuie evitat întrucât devine neclar cititorului unde este definiţia acelei variabile. Folosind declaraţiaglobal marcăm foarte clar că variabila este definită în cel mai exterior bloc.
Exemplu:

```
#!/usr/bin/python
# Fişier: func_global.py

x = 50

def func():
     global x

     print('x is', x)
     x = 2
     print('Am schimbat x global în ', x)

func()
print('Valoarea lui x este', x)
```

Rezultat:

```
   $ python func_global.py

   x este 50

   Am schimbat x global în 2

   Valoarea lui x este 2
```

Cum funcţionează:

Declaraţia global este folosită pentru a declara că x este o variabilă globală – de aceea, când atribuim o valoare lui x în interiorul funcţiei, acea schimbare se reflectă când folosim valoarea lui x în blocul principal.
Puteţi specifica mai multe variabile globale folosind declaraţia global. De exemplu, global x, y, z.

Folosirea declaraţiei nonlocal

Am învăţat să accesăm variabile în domeniul local şi global. Mai există un domeniu specific funcţiilor, numit “nonlocal” şi care se află între cele două. Domeniile nonlocal se observă când definiţi funcţii în interiorul funcţiilor.

Întrucât totul în Python este cod executabil, se pot defini funcţii oriunde.

Să luăm un exemplu:

```
#!/usr/bin/python
# Fişier: func_nonlocal.py

def func_outer():
    x = 2
    print('x este', x)

    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('x local a devenit ', x)

func_outer()
```

Rezultat:

```
$ python func_nonlocal.py


x este 2


x local a devenit 5
```

Cum funcţionează:

Când ne aflăm în interiorul unei funcţii func_inner, ‘x’ definit în prima linie a funcţieifunc_outer este undeva între global şi local. Declarăm că folosim acest x cu declaraţianonlocal x şi astfel obţinem acces la acea variabilă.
Încercaţi să schimbaţi nonlocal x cu global x şi să eliminaţi complet declaraţia, ca să vedeţi ce diferenţe de comportament sunt în aceste cazuri.

Valori implicite ale argumentelor

Pentru unele funcţii, poate vreţi să faceţi unii parametri *opţionali* şi să folosiţi valori implicite în cazul în care utilizatorul nu furnizează o valoare pentru parametrul respectiv. Asta se face cu ajutorul valorilor implicite ale parametrilor. Puteţi specifica valorile implicite ale argumentelor în definiţia funcţiei, punând operatorul de atribuire (=) urmat de valoarea implicită.
Observaţi că valoarea implicită a argumentului trebuie să fie o constantă. Mai precis, trebuie să fie imuabilă – acest fapt va fi explicat în detaliu în capitolele următoare. Pentru moment reţineţi doar atât.

Exemplu:

```
#!/usr/bin/python
# Fişier: func_default.py

def say(mesaj, ori = 1):
    print(mesaj * ori)
```

```
say('Hello')
say('World', 5)
```

Rezultat:

```
$ python func_default.py

Hello

WorldWorldWorldWorldWorld
```

Cum funcţionează:

Funcţia numită say este folosită pentru a tipări pe ecran un şir de atâtea ori cât se specifică. Dacă nu furnizăm acea valoare, atunci va fi folosită valoarea implicită, 1. Obţinem aceasta punând valoarea implicită 1 a parametrului ori.
La prima folosire a funcţiei say, dăm numai şirul şi ea îl tipăreşte o dată. În a doua folosire dăm funcţiei say şi şirul şi un argument 5 ceea ce spune că vrem să fie tipărit şirul de 5 ori.

Important

Numai parametrii de la sfârşitul listei de parametri pot avea valori implicite deci nu puteţi avea un parametru cu valoare implicită înaintea altuia fără valoare implicită în lista de parametri a funcţiei.

Motivul este că valorile sunt atribuite parametrilor prin poziţie. De exemplu, def func(a, b=5) este validă, dar def func(a=5, b) este *invalidă*.

Argumente cuvânt cheie

Dacă aveţi funcţii cu mulţi parametri şi vreţi să specificaţi numai pe unii, atunci puteţi să daţi valori parametrilor prin numele lor – acest mod de specificare se numeşte *prin argumente cuvânt cheie* – folosim cuvântul cheie (engl. keyword) în locul poziţiei (pe care am folosit-o până acum) pentru a specifica argumente funcţiei.

Există două *avantaje* – unu, folosirea funcţiei este mai uşoară, întrucât nu trebuie să ne preocupăm de ordinea parametrilor. Doi, putem da valori numai unor parametri selectaţi, cu condiţia ca toţi ceilalţi sa aibă în definiţia funcţiei valori implicite.

Exemplu:

```
#!/usr/bin/python
# Fişier: func_key.py

def func(a, b=5, c=10):
```

```
    print('a este', a, 'şi b este', b, 'şi c este', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Rezultat:

```
$ python func_key.py

a este 3 şi b este 7 şi c este 10

a este 25 şi b este 5 şi c este 24

a este 100 şi b este 5 şi c este 50
```

Cum funcţionează:

Funcţia numită func are un parametru fără valoare implicită, urmat de doi parametri cu valori implicite.
În primul apel, func(3, 7), parametrul a primeşte valoarea 3, parametrul b primeşte valoarea 7, iar c valoarea implicită, 10.
În al doilea apel, func(25, c=24), variabila a ia valoarea 25 datorită poziţiei argumentului. Pe urmă, parametrul c ia valoarea 24 prin nume – argument cuvânt cheie. Variabila b ia valoarea implicită, 5.
În al treilea apel, func(c=50, a=100), folosim numai tehnica nouă, a cuvintelor cheie. Observaţi, specificăm valoarea parametrului c înaintea parametrului a deşi a este definit înaintea variabilei c în definiţia funcţiei.

Parametri VarArgs

TODO

> Să scriu despre asta într-un capitol următor, fiindcă nu am vorbit încă despre liste şi dicţionare?

Uneori aţi putea dori să definiţi o funcţie care să ia *orice* număr de parametri, asta se poate face folosind asteriscul:

```
#!/usr/bin/python
# Fişier: total.py

def total(iniţial=5, *numere, **keywords):
    numărător = iniţial
    for număr in numere:
        numărător += număr
    for cheie in keywords:
```

```
        numărător += keywords[cheie]
    return numărător

print(total(10, 1, 2, 3, legume=50, fructe=100))
```

Rezultat:

```
$ python total.py

166
```

Cum funcţionează:

Când declarăm un parametru cu asterisc precum *parametri, toţi parametrii poziţionali de la acel punct încolo sunt colectaţi întro listă numită ‘parametri’.
Similar, când declarăm un parametru cu două asteriscuri, precum **parametri, toate argumentele cuvânt cheie de la acel punct încolo sunt colectate într-un dicţionar numit ‘parametri’.
Vom explora listele şi dicţionarele întrun capitol următor.

Parametri exclusiv cuvânt cheie

Dacă vrem să specificăm anumiţi parametri cuvânt cheie pentru a fi disponibili numai în forma cuvânt cheie şi *niciodată* ca parametri poziţionali, aceştia pot fi declaraţi după un parametru cu asterisc:

```
#!/usr/bin/python
# Fişier: keyword_only.py

def total(iniţial=5, *numere, legume):
    numărător = iniţial
    for număr in numere:
        numărător += număr
    numărător += legume
    return numărător

print(total(10, 1, 2, 3, legume=50))
print(total(10, 1, 2, 3))
# Ridică o eroare pentru că nu am furnizat o valoare implicită pentru 'legume'
```

Rezultat:

```
$ python keyword_only.py

66

Traceback (most recent call last):

  File "test.py", line 12, in <module>

print(total(10, 1, 2, 3))
```

```
TypeError: total() needs keyword-only argument legume
```

Cum funcţionează:

Declararea de parametri după un parametru cu asterisc (engl. starred parameter) produce argumente exclusiv cuvânt cheie. Dacă acestea nu sunt definite cu valori implicite, apelurile funcţiei vor ridica o eroare dacă nu se furnizează argumentul cuvânt cheie, aşa cum s-a văzut mai sus.

Dacă vreţi să aveţi parametri exclusiv cuvânt cheie, dar nu aveţi nevoie de nici un parametru cu asterisc, folosiţi un asterisc izolat, ca în exemplul:

def total(iniţial=5, *, legume).

Declaraţia return

Declaraţia return este folosită pentru a ne *întoarce* dintr-o funcţie (deci a evada din ea – engl. break out). Opţional putem *întoarce o valoare* la fel de bine.

Exemplu:

```
#!/usr/bin/python
# Fişier: func_return.py

def maximum(x, y):
    if x > y:
        return x
    else:
        return y

print(maximum(2, 3))
```

Rezultat:

```
$ python func_return.py

3
```

Cum funcţionează:

Funcţia maximum întoarce parametrul cel mai mare furnizat funcţiei. Ea foloseşte o declaraţie simplă if..else pentru a găsi numărul cel mai mare şi apoi *întoarce* (engl. return) acea valoare. Observaţi că declaraţia return fără o valoare este echivalentă cu return None. None este un tip special în Python care reprezintă nimicul. De exemplu, este folosit pentru a indica faptul că o variabilă nu are nici o valoare, deci are valoarea None.

Orice funcţie, în mod implicit, conţine o declaraţie return None la sfârşitul blocului de declaraţii, cu excepţia cazului în care îi scrieţi o altă declaraţie return. Puteţi vedea asta rulând print o_funcţie_oarecare() în care nu este dată o declaraţie return precum:

```
def o_functie_oarecare():
    pass
```

Declaraţia pass este folosită în Python pentru a indica un bloc de declaraţii gol.

Notă

Există o funcţie predefinită numită max care implementează această funcţionalitate de a ‘găsi maximul’, deci folosirea aceste funcţii este posibilă oricand.

DocStrings

Python are o facilitate drăguţă numită *documentation strings*, numită de obicei pe numele scurt *docstrings*. DocStrings (rom. şiruri de documentaţie, sg. docstring) sunt o unealtă importantă pentru că vă ajută să documentaţi programele mai bine şi le face mai uşor de înţeles. Uimitor, putem chiar să extragem şirurile de documentare ale, să zicem, unei funcţii chiar în timp ce programul rulează!

Exemplu:

```
#!/usr/bin/python
# Fişier: func_doc.py

def printMax(x, y):
    '''Tipăreşte pe ecran cel mai mare din două numere.

  Cele două numere trebuie să fie întregi.'''
    x = int(x)  # converteşte în integer, dacă este posibil
    y = int(y)

    if x > y:
        print(x, 'este maximum')
    else:
        print(y, 'este maximum')

print(printMax.__doc__)
printMax(3, 5)
```

Rezultat:

```
$ python func_doc.py

Tipăreşte pe ecran cel mai mare din două numere.



    Cele două numere trebuie să fie întregi.
```

```
5 este maximum
```

Cum funcţionează:

Un şir pe prima linie logică a funcţiei devine *docstring* pentru acea funcţie. De retinut că DocStrings se aplică şi la module şi clase, despre care vom învăţa în capitolele respective. Convenţia urmată pentru un docstring este: un şir multilinie în care prima linie începe cu majusculă şi se încheie cu punct. Apoi linia a doua este goală şi urmată de o explicaţie mai detaliată începand cu linia a treia. Vă *sfătuim cu căldură* să urmaţi aceasta convenţie pentru toate docstringurile tuturor funcţiilor nebanale pe care le scrieţi.

Putem accesa docstringul funcţiei printMax folosind atributul __doc__ (observaţi *dublu underscore*) al funcţiei. Amintiţi-vă că Python tratează *totul* ca obiect, inclusiv funcţiile. Vom învăţa mai mult despre obiecte în capitolul despre clase.
Daca aţi folosit help() în Python, aţi văzut deja cum se foloseşte docstring! Ceea ce face ea este că extrage atributul __doc__ al funcţiei şi îl afişează într-o maniera convenabilă. Puteţi încerca asta asupra funcţiei de mai sus – includeţi pur şi simplu declaraţiahelp(printMax) în program. Nu uitaţi să tastaţi q pentru a ieşi din help.
Utilitarele pot colecta automat documentaţia din programe în această maniera. De aceea*vă recomand insistent* să folosiţi docstring pentru orice funcţie nebanală pe care o scrieţi.
Comanda pydoc inclusă în distribuţia Python funcţionează similar cu help() folosind docstringurile.

Adnotări

Funcţiile mai au o facilitate avansată numită adnotare (engl. annotations) care este o cale deşteaptă de a ataşa informaţie pentru fiecare din parametri precum şi pentru valoarea întoarsă. Întrucât limbajul Python în sine nu interpretează aceste adnotări în nici un fel (această funcţionalitate este lăsată bibliotecilor third-party să interpreteze ele în ce fel vor), vom trece peste această facilitate în discuţia noastră. Dacă sunteţi interesaţi despre adnotări, puteţi citi PEP No. 3107.

Rezumat

Am discutat multe aspecte ale funcţiilor, dar reţineţi că nu am acoperit toate aspectele posibile. Totuşi, am acoperit deja majoritatea aspectelor pe care le vom folosi în mod uzual.

Vom afla în continuare cum să folosim, dar şi să cream module Python.

Python ro:Module

Contents

Introducere

Aţi văzut cum se poate refolosi o porţiune de cod în program prin definirea funcţiilor. Dar dacă vreţi să refolosiţi un număr mai mare de funcţii în alte programe decât cel pe care îl scrieţi? Aşa cum aţi ghicit, răspunsul este folosirea modulelor.

Există variate metode de a scrie module, dar cea mai simplă cale este de a crea un fişier cu extensia .py care conţine funcţii şi variabile.
Altă metodă este scrierea modulelor în limbajul în care chiar interpretorul Python a fost scris. De exemplu, puteţi scrie module în limbajul de programare C şi dupa compilare, ele pot fi folosite din codul Python când se foloseşte interpretorul Python standard.
Un modul poate fi *importat* de un alt program pentru a folosi funcţionalitatea acestuia. Aşa putem şi noi să folosim biblioteca standard Python. Întâi vom vedea cum se folosesc modulele bibliotecii standard.

Exemplu:

```
#!/usr/bin/python
# Fişier: using_sys.py

import sys

print('Argumentele la linia de comandă sunt:')
for i in sys.argv:
    print(i)

print('\n\nPYTHONPATH este', sys.path, '\n')
```

Rezultat:

```
$ python using_sys.py noi suntem argumente

Argumentele la linia de comandă sunt:

using_sys.py

noi
```

```
   suntem

   argumente



   PYTHONPATH este ['', 'C:\\tmp', 'C:\\Python30\\python30.zip',

   'C:\\Python30\\DLLs', 'C:\\Python30\\lib', 'C:\\Python30\\lib\\plat-
win',

   'C:\\Python30', 'C:\\Python30\\lib\\site-packages']
```

Cum funcţionează:

La început *importăm* modulul sys folosind declaraţia import. În esenţă, asta îi spune lui Python că vrem să folosim acest modul. Modulul sys conţine funcţionalitate legată de interpretorul Python şi mediul său, *sys*tem.
Când Python execută declaraţia import sys, el caută modulul sys. În acest caz, este vorba de un modul preinstalat şi de aceea Python ştie unde să-l găsească.
Dacă nu ar fi fost un modul compilat, ci un modul scris în Python, interpretorul ar fi căutat în directoarele listate în variabila sys.path. Dacă modulul este găsit, declaraţiile din interiorul modului sunt executate. Observaţi că această iniţializare este făcută numai*prima* dată când importăm un modul.
Variabila argv din modulul sys este accesată folosind notaţia cu puncte, adică sys.argv. Ea arată clar că acest nume este parte a modulului sys. Alt avantaj al acestei abordări este că numele nu dă conflict cu nici o variabilă argv folosită în program.
Variabila sys.argv este o *listă* de şiruri (listele sunt explicate în detaliu în capitolul despre liste. În special, variabila sys.argv conţine lista *argumentelor din linia de comandă* adică acele argumente transmise programului prin adăugarea lor la linia de comandă care lansează programul.
Daca folosiţi un IDE pentru a scrie şi rula aceste programe, căutaţi în meniuri o cale de a specifica argumente la linia de comandă.

Aici, când se execută python using_sys.py noi suntem argumente, rulăm modulului using_sys.py cu comanda python şi celelalte lucruri care îl urmează sunt transmise programului. Python păstrează linia de comandă în variabila sys.argv ca să le putem folosi.
Reţineţi, numele scriptului care rulează este întotdeauna primul argument din listasys.argv. Deci în acest caz vom avea 'using_sys.py' în poziţia sys.argv[0], 'noi' în poziţiasys.argv[1], 'suntem' în poziţia sys.argv[2] şi 'argumente' în poziţia sys.argv[3]. Observaţi că Python începe numerotarea cu 0 nu cu 1.

Variabila sys.path conţine lista numelor de director de unde pot fi importate module. Observaţi că primul sir din sys.path este vid – asta arată că directorul curent este parte a variabilei sys.path ceea ce este totuna cu variabila de mediu PYTHONPATH. Acest comportament este prevăzut pentru a permite importul direct al modulelor aflate în directorul curent. În caz contrar modulele care trebuie importate trebuie poziţionate într-unul din directoarele listate în sys.path.

Fisiere .pyc compilate in octeti

Importul unui modul este relativ costisitor, astfel că Python face nişte smecherii ca să îl accelereze. O cale este să creeze fişiere *compilate în octeţi* (engl. byte-compiled) cu extensia .pyc care sunt nişte forme intermediare în care Python transformă programul (vă amintiţi din capitolul introductiv cum lucrează Python?). Acest fişier .pyc este util când importaţi un modul a doua oară din alte programe – ele vor fi mult mai rapide întrucât partea de procesare legată de importul modulului este deja realizată. De asemenea, aceste fişiere compilate în octeţi sunt independente de platformă.

Notă

Fişierele .pyc sunt create de obicei în acelaşi director ca şi fişierul .pycorespondent. Dacă Python nu are permisiunea de a scrie fişiere în acel director, fişierele .pyc nu vor fi create.

Declaraţia from … import …

Dacă vreţi să importaţi direct variabila argv în programul vostru (pentru a evita scrierea numelui sys. de fiecare dată), puteţi folosi declaraţia from sys import argv. Dacă vreţi să importaţi toate numele folosite în modulul sys atunci puteţi folosi declaraţia from sys import *. Funcţionează pentru orice modul.

În general, *trebuie să evitaţi* folosirea acestor declaraţii şi în schimb să folosiţi declaraţiaimport. Ca să fie evitate orice conflicte de nume şi programele să fie mai lizibile.

Atributul __name__ al modulului

Orice modul are un nume, iar declaraţiile din modul pot găsi numele modulului. Este comod aşa, mai ales în situaţia particulară în care se doreşte aflarea regimului modulului (autonom sau importat). Cum am menţionat anterior, când un modul este importat pentru prima dată, codul din modul este executat. Putem folosi acest concept pentru a altera comportamentul modulului dacă programul este executat autonom şi îl putem lăsa neschimbat dacă modulul este importat din alt modul. Acestea sunt posibile folosind atributul __name__ al modulului.

Exemplu:

```
#!/usr/bin/python
# Fişier: using_name.py

if __name__ == '__main__':
    print('Acest program rulează autonom')
else:
    print('Acest program a fost importat din alt modul')
```

Rezultat:

```
$ python using_name.py

Acest program rulează autonom



$ python

>>> import using_name

Acest program a fost importat din alt modul

>>>
```

Cum funcţionează:

Orice modul Python are propriul atribut __name__ definit şi dacăş acesta este '__main__', rezultă că acel modul este rulat de sine stătător de către utilizator şi putem lua măsurile adecvate.

Crearea propriilor module

Crearea propriilor noastre module este uşoară, aţi făcut asta tot timpul! Asta din cauză că orice program Python este un modul. Trebuie doar să ne asigurăm că fişierul are extensia.py. Următorul exemplu ar trebui să clarifice situaţia.

Exemplu:

```
#!/usr/bin/python
# Fişier: meu.py

def zisalut():
      print('Salut, aici este modulul meu.')

__versiune__ = '0.1'

# Sfârşitul modulului meu.py
```

Mai sus a fost un model de *modul*. Aşa cum vedeţi, nu este nimic deosebit în legătură cu modulele în comparaţie cu programele Python obişnuite. Vom vedea în continuare cum putem să folosim acest modul în programele noastre Python.

Amintiţi-vă că modulul ar trebui plasat în acelaşi director cu programul care îl importă sau într-un director listat în variabila sys.path.

```
#!/usr/bin/python
# Fişier: meu_demo.py

import meu
```

```
meu.zisalut()
print ('Versiunea', meu.__versiune__)
```

Rezultat:

```
$ python meu_demo.py

Salut, aici este modulul meu.

Versiunea 0.1
```

Cum funcţionează:

Observaţi că folosim aceeaşi notaţie cu punct pentru a accesa membrii modulului. Python refoloseşte cu spor aceeaşi notaţie pentru a da un sentiment distinctiv 'Pythonic' programelor, astfel încât să nu fim nevoiţi să învăţăm noi moduri de a face lucrurile.

Iată o nouă variantă folosind sintaxa declaraţiei from..import:

```
#!/usr/bin/python
# Fişier: meu_demo2.py

from meu import zisalut, __versiune__

zisalut()
print('Versiunea', __versiune__)
```

Rezultatul programului meu_demo2.py este acelaşi ca şi rezultatul programuluimeu_demo.py. Observaţi că dacă ar fi existat un nume __versiune__ declarat în modulul care importă modulul meu, ar fi apărut un conflict. Şi asta este probabil, întrucât este o practică uzuală pentru fiecare modul să se declare versiunea sa folosind acest nume. De aceea este întotdeauna recomandabil să se folosească declaraţia import deşi ar putea face programul un pic mai lung.
S-ar mai putea folosi:

```
from meu import *
```

Astfel ar fi importate toate numele publice, precum zisalut dar nu s-ar importa__versiune__ pentru ca începe cu dublu underscore.

Calea (Zen) în Python

> Un principiu director în Python este "Explicit este mai bine decât implicit".
> Rulaţiimport this pentru a afla mai multe şi urmăriţi această discuţie care enumeră exemple pentru fiecare din principii.

Funcţia dir

Puteţi folosi funcţia predefinită dir pentru a lista identificatorii pe care îi defineşte un obiect. De exemplu, pentru un modul, clasele şi variabilele definite în acel modul.

Când furnizaţi un nume funcţiei dir(), ea întoarce lista numelor definite în acel modul. Dacă se lansează funcţia fără argumente, ea întoarce lista numelor definite în modulul curent.
Exemplu:

```
$ python




>>> import sys # obţine lista atributelor, în acest caz, pentru modulul sys

>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__', '__s
tderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_compact_freelists',
'_current_frames', '_getframe', 'api_version', 'argv', 'builtin_module_names', '
byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle'
, 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding', 'getfil
esystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
 'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize', 'maxunicode
', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platfor
m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit
', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_in
fo', 'warnoptions', 'winver']

>>> dir() # obţine lista atributelor pentru modulul curent['__builtins__', '__doc__', '__name__', '__package__',
'sys']

>>> a = 5 # crează o nouă variabilă, 'a'

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']

>>> del a # şterge (engl. delete) un nume

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']

>>>
```

Cum funcţionează:

Pentru început, vedem folosirea funcţiei dir asupra modulului importat sys. Putem vedea lista uriaşă de atribute pe care o conţine.
Apoi folosim dir fără parametri. Implicit, ea întoarce lista atributelor modulului curent. Observaţi că lista modulelor importate este inclusă în lista modulului listat.
Pentru a vedea funcţia dir în acţiune, definim o nouă variabilă, a, şi îi atribuim o valoare, apoi testăm cu dir dacă a apărut încă o valoare în lista de atribute a aceluiaşi nume. Eliminăm variabila/atributul modulului curent folosind declaraţia del şi din nou schimbarea este reflectată în rezultatul funcţiei dir.

O notă asupra declaraţiei del – această declaraţie este folosită pentru a *şterge* un nume de variabila şi după ce a fost executată (del a), nu mai puteţi accesa variabila a – este ca şi cum nu a existat niciodată.
Reţineţi că funcţia dir() lucrează pe *orice* obiect. De exemplu, rulaţi dir(print) pentru a descoperi atributele funcţiei print sau dir(str) pentru atributele clasei str.

Pachete

La acest nivel, aţi început probabil să observaţi o ierarhie în organizare a programelor. Variabilele sunt de obicei în interiorul funcţiilor. Funcţiile şi variabilele globale intră în module. Dar modulele cum se organizează? Aici intervin pachetele.

Pachetele sunt nişte foldere de module cu un fişier special __init__.py care indică lui Python că acel folder este special, deoarece conţine module Python.
Să zicem că vreţi să creaţi un pachet numit ‘mapamond’ cu subpachetele ‘asia’, ‘africa’, etc. şi aceste pachete conţin la rândul lor module precum ‘india’, ‘madagascar’, ‘românia’ etc.

Iată cum aţi structura folderele:

```
- <un folder prezent în sys.path>/

    - mapamond/

        - __init__.py

        - asia/

            - __init__.py

            - india/

                - __init__.py

                - foo.py

        - africa/

            - __init__.py

            - madagascar/

                - __init__.py

                - bar.py
```

```
        - europa/
            - __init__.py
            - românia/
                - __init__.py
                - foo_bar.py
```

Pachetele sunt doar un mod convenabil de a organiza ierarhic modulele. Veţi vedea de multe ori asta în biblioteca Python standard.

Rezumat

Aşa cum funcţiile sunt părţi reutilizabile de program, modulele sunt programe (întregi) reutilizabile. O altă ierarhie de organizare a modulelor o reprezintă pachetele. Biblioteca standard care vine cu Python este un exemplu de set de pachete şi module.

Am văzut cum se folosesc modulele şi cum se creeaza module proprii.

În continuare vom învăţa despre câteva concepte interesante numite ‘structuri de date’.

Python ro:Structuri de date

Contents

Introducere

Structurile de date sunt în esenţă exact asta – *structuri* care pot memora *date* grupate. Cu alte cuvinte, sunt folosite pentru a stoca colecţii de date înrudite.

Există patru structuri predefinite în Python – liste, tupluri, dicţionare şi seturi. Vom învăţa să le folosim pe fiecare şi cum ne pot uşura ele viaţa.

Liste

O listă (engl. list) este o structură de date care păstrează o colecţie ordonată de elemente deci se poate memora o *secvenţă* de elemente întro listă. Asta este uşor de imaginat dacă ne gândim la o listă de cumpărături, numai că pe listă fiecare item ocupă un rând separat, iar în Python punem virgule intre ele.

Elementele listei trebuie incluse în paranteze drepte astfel încât Python să înţeleagă că este o specificare de listă. Odată ce am creat lista, putem adăuga, şterge sau căuta elemente în ea. De aceea se poate spune că listele sunt *muabile*, acest tip de date poate fi modificat.

INTRODUCERE RAPIDĂ ÎN OBIECTE ŞI CLASE

Deşi în general am amânat discutarea obiectelor şi claselor pâna acum, este necesară o scurtă introducere, pentru a se putea înţelege listele mai bine. Detaliile le vom afla în capitolul dedicat acestora.

O listă este un exemplu de utilizare a obiectelor şi claselor. Când folosim o variabilă i şi îi atribuim o valoare, să zicem întregul 5, putem gândi că am creat un **obiect** (de fapt instanţă) i din **clasa** (de fapt tipul) int. De fapt se poate citi help(int) pentru o înţelegere mai aprofundatată.

O clasă poate avea şi **metode** adică funcţii definite pentru a fi folosite exclusiv în raport cu acea clasă. Puteţi folosi aceste funcţionalităţi numai asupra unui obiect din acea clasă. De exemplu, Python oferă o metodă append pentru clasa list care ne permite să adăugăm un element la sfârşitul listei. Prin urmare lista_mea.append('un item') va adăuga acel şir lalista_mea. De reţinut folosirea notaţiei cu punct pentru accesarea metodelor obiectelor.

O clasă poate avea şi **câmpuri** (engl. fields) care nu sunt altceva decât variabile definite în raport exclusiv cu acea clasă. Se pot folosi acele variabile/nume numai în raport un obiect din acea clasă. Câmpurile sunt accesate tot prin notaţia cu punct, de exemplu lista.câmp.

Exemplu:

```
#!/usr/bin/python
# Fişier: using_list.py

# Lista mea de cumpărături
shoplist = ['mere', 'mango', 'morcovi', 'banane']

print('Am de cumpărat', len(shoplist), 'itemuri.')

print('Acestea sunt:', end=' ')
for item in shoplist:
    print(item, end=' ')

print('\nTrebuie să cumpăr şi orez.')
shoplist.append('orez')
print('Lista mea de cumpărături este acum', shoplist)
```

```
print('Acum vom sorta lista')
shoplist.sort()
print('Lista sortată este', shoplist)

print('Primul lucru de cumpărat este', shoplist[0])
item_cumpărat = shoplist[0]
del shoplist[0]
print('Am cumpărat', item_cumpărat)
print('Lista mea este acum', shoplist)
```

Rezultat:

```
$ python using_list.py

Am de cumpărat 4 itemuri.

Acestea sunt: mere mango morcovi banane

Trebuie să cumpăr şi orez.

Lista mea de cumpărături este acum

['mere', 'mango', 'morcovi', 'banane', 'orez']

Acum vom sorta lista

Lista sortată este

['banane', 'mango', 'mere', 'morcovi', 'orez']

Primul lucru de cumpărat este banane

Am cumpărat banane

Lista mea este acum

['mango', 'mere', 'morcovi', 'orez']
```

Cum funcţionează:

Variabila shoplist este o listă de cumpărături pentru cineva care merge la piaţă. În shoplist, memorăm şiruri care reprezintă numele lucrurilor pe care le avem de cumpărat, dar putem adăuga *orice fel de obiect* inclusiv numere sau alte liste.
Am folosit bucla for..in pentru a itera prin itemurile listei. Până acum cred că aţi realizat ca o listă este şi secvenţă în acelaşi timp. Specificul secvenţelor va fi discutat întrunsubcapitol următor.

Observaţi utilizarea argumentului cuvânt cheie end al funcţiei print pentru a-i transmite că vrem să încheiem linia cu un spaţiu (‘ ‘) în loc de încheierea uzuală.
În continuare adăugăm un item la listă folosind metoda append a obiectului listă, aşa cum am discutat anterior. Verificăm că adăugarea s-a realizat tipărind conţinutul listei prin transmiterea ei funcţiei print care o tipăreşte frumos pe ecran.
Mai departe sortăm lista folosind metoda sort a listei. Este important să înţelegem că această metodă afecteaza însăşi lista şi nu întoarce o listă modificată – spre deosebire de comportamentul sirurilor. Asta vrem să zicem prin *muabile* pe când şirurile sunt *imuabile*.
După ce cumpărăm un item de la piaţă, vrem să-l eliminăm din listă. Pentru aceasta utilizăm declaraţia del. Trebuie menţionat aici itemul pe care vrem să-l eliminăm şi declaraţia del îl elimină pentru noi. Specificăm că vrem să eliminăm primul item, de aici declaraţia del shoplist[0] (amintiţi-vă că Python începe numărătoarea de la 0).
Dacă vreţi să ştiţi toate metodele definite de obiectul listă, citiţi help(list).

Tupluri

Tuplurile sunt folosite pentru păstra colecţii de obiecte. Sunt similare cu listele, dar fără funcţionalitatea extinsă pe care o dau clasele. O facilitate majoră a tuplurilor este că ele sunt **imuabile** ca şi şirurile adică nu pot fi modificate.

Tuplurile sunt definite prin specificarea itemurilor separate prin virgule întro pereche opţională de paranteze.

Tuplurile sunt folosite de obicei în cazurile în care o declaraţie sau o funcţie definită de utilizator poate presupune fără risc de greşeală că o colecţie de valori nu se va schimba..

Exemplu:

```
#!/usr/bin/python
# Fişier: using_tuple.py

zoo = ('piton', 'elefant', 'pinguin')  # reţineţi că parantezele sunt opţionale
print('Numărul animalelor în zoo este', len(zoo))

zoo_nou = ('maimuţă', 'cămilă', zoo)
print('Numărul de cuşti în noul zoo este', len(zoo_nou))
print('Animalele din noul zoo sunt ', zoo_nou)
print('Animalele aduse din vechiul zoo sunt ', zoo_nou[2])
print('Ultimul animal adus din vechiul zoo este', zoo_nou[2][2])
print('Numărul de animale în noul zoo este', len(zoo_nou)-1+len(zoo_nou[2]))
```

Rezultat:

```
$ python using_tuple.py

Numărul animalelor în zoo este 3
```

```
   Numărul de cuşti în noul zoo este 3

   Animalele din noul zoo sunt ('maimuţă', 'cămilă', ('piton', 'elefant',
'pinguin'))

   Animalele aduse din vechiul zoo sunt ('piton', 'elefant', 'pinguin')

   Ultimul animal adus din vechiul zoo este pinguin

   Numărul de animale în noul zoo este 5
```

Cum funcţionează:

Variabila zoo este un tuplu de itemuri. Vedem că funcţia len lucrează şi pentru tupluri. De asemenea asta arată că tuplurile sunt şi secvenţe.
Acum mutăm aceste animale într-un nou zoo, deoarece vechiul zoo s-a închis, să zicem. Ca urmare tuplul zoo_nou conţine nişte animale care erau acolo împreună cu animalele aduse din vechiul zoo. În realitate, acum, reţineţi că un tuplu în interiorul altui tuplu nu îşi pierde identitatea. Putem accesa itemurile din tuplu specificând poziţia întro pereche de paranteze pătrate, ca pentru liste. Acesta se numeşte operator de *indexare*. Accesăm al treilea item dinzoo_nou specificând zoo_nou[2] şi accesăm al treilea item al tuplului zoo din tuplulzoo_nou specificând zoo_nou[2][2]. E destul de simplu după ce aţi înteles regula.

Paranteze

Deşi parantezele sunt opţionale, eu prefer să le pun mereu, pentru a face evident că e vorba de un tuplu, în special pentru a evita ambiguitatea. De exempluprint(1,2,3) şi print((1,2,3)) înseamnă două lucruri foarte diferite – prima tipăreşte trei numere, iar a doua tipăreşte un tuplu (care conţine trei numere).

Tupluri cu 1 sau 0 elemente

Un tuplu vid este construit folosind o pereche de paranteze goale myempty = ().
Totuşi, un tuplu cu un singur element nu e aşa de simplu. Trebuie să îl specificaţi folosind virgula după primul (şi ultimul) element, ca Python să poată diferenţia între tuplu şi un alt obiect cuprins în paranteze întro expresie deci va trebui să specificaţi singleton = (2 ,) dacă vreţi să se înţeleagă ‘tuplul care conţine doar elementul 2‘.

Notă pentru programatorii în Perl

O listă întro listă nu-şi pierde identitatea adică nu este asimilată ca în Perl. Asta se aplică şi pentru un tuplu întrun tuplu, o listă întrun tuplu, un tuplu întro listă etc. În ce priveşte limbajul Python, ele sunt nişte obiecte stocate în alte obiecte.

Dicţionare

Un dicţionar este ca o carte de adrese în care poţi găsi adresa sau datele de contact ale persoanei doar ştiindu-i numele, adică asociem **chei** (nume) cu **valori** (detalii). De observat că o cheie trebuie să fie unică, pentru a nu exista confuzii, exact ca atunci când nu poţi deosebi două persoane dacă au acelaşi nume.

Pe post de chei puteţi folosi numai obiecte imuabile (precum şirurile), dar pe post de valori putem folosi orice fel de valori. În esenţă înseamnă că ar trebui să folosim pe post de chei numai obiecte simple.

Perechile cheie – valoare sunt specificate întrun dicţionar folosind notaţia d = {cheie1: valoare1, cheie2: valoare2 }. Observaţi că perechile cheie – valoare sunt separate prin virgulă, iar cheia este separată de valoare prin semnul două puncte. Dicţionarul este delimitat de o pereche de acolade. Reţineţi că întrun dicţionar perechile cheie – valoare nu sunt ordonate în nici un fel. Dacă vreţi o anumită ordine, va trebui să îl sortaţi singuri înainte de folosire.

Dicţionarele pe care le veţi folosi sunt instanţe/obiecte ale clasei dict.
Exemplu:

```
#!/usr/bin/python
# Fişier: using_dict.py

# 'ab' este o prescurtare de la 'a'ddress'b'ook

ab = {   'Swaroop'   : 'swaroop@swaroopch.com',
         'Larry'     : 'larry@wall.org',
         'Matsumoto': 'matz@ruby-lang.org',
         'Spammer'   : 'spammer@hotmail.com'
     }

print("Adresa lui Swaroop este", ab['Swaroop'])

# Ştergerea unei perechi cheie - valoare
del ab['Spammer']

print('\nExistă {0} contacte în address-book\n'.format(len(ab)))

for nume, adresa in ab.items():
     print('Contactaţi pe {0} la adresa {1}'.format(nume, adresa))

# Adăugarea unei perechi cheie - valoare
ab['Guido'] = 'guido@python.org'
```

```
if 'Guido' in ab: # OR ab.has_key('Guido')
    print("\nAdresa lui Guido este", ab['Guido'])
```

Rezultat:

```
$ python using_dict.py

Adresa lui Swaroop este swaroop@swaroopch.com



Există 3 contacte în address-book



Contactaţi pe Swaroop la adresa swaroop@swaroopch.com

Contactaţi pe Matsumoto la adresa matz@ruby-lang.org

Contactaţi pe Larry la adresa larry@wall.org



Adresa lui Guido este guido@python.org
```

Cum funcţionează:

Creăm dicţionarul ab folosind notaţia deja discutată. Accesăm perechile cheie – valoare specificând cheia şi folosind operatorul de indexare, aşa cum am discutat la liste şi tupluri. Observaţi simplitatea sintaxei.

Putem şterge perechi cheie valoare folosind vechiul nostru prieten – declaraţia del. Pur şi simplu specificăm dicţionarul şi operatorul de indexare pentru cheia care trebuie ştearsă şi le dăm declaraţiei del. Nu este necesar să se cunoască şi valoarea asociata cheii pentru a realiza această operaţie.

În continuare accesăm fiecare pereche cheie – valoare din dicţionar folosind metodaitems a dicţionarului care întoarce o listă de tupluri în care fiecare tuplu conţine o pereche de itemuri – cheia urmată de valoare. Extragem această pereche şi o atribuim variabilelornume şi adresă corespunzător pentru fiecare pereche folosind o buclă for..in în care tipărim aceste informaţii.

Putem adăuga perechi noi cheie – valoare prin simpla utilizare a operatorului de indexare pentru a accesa cheia şi a atribui acea valoare, aşa cum am făcut pentru Guido în cazul de mai sus.

Putem testa dacă există în dicţionar o anumită pereche cheie – valoare folosind operatorulin sau chiar metoda has_key a clasei dict. Puteţi consulta documentaţia pentru o listă completă a metodelor clasei dict folosind comanda help(dict).

Argumente cuvânt cheie şi dicţionare

Într-o altă ordine de idei, dacă aţi folosit argumente cuvânt cheie în funcţii, înseamnă că aţi folosit deja dicţionare! Ia gândiţi-vă: perechea cheie – valoare este specificată în lista de parametri a definiţiei funcţiei şi când accesaţi variabilele, numele lor sunt chei de acces ale unui dicţionar numit *tabel de simboluri* în terminologia proiectării de compilatoare).

Secvenţe

Listele, tuplurile şi şirurile sunt exemple de secvenţe, dar ce sunt secvenţele şi ce le face atât de speciale?

Facilitatea cea mai importantă este că au teste de apartenenţă (adică expresiile in şi not in) şi operaţii de indexare. Operaţia de **indexare** ne permite să extragem direct un item din secvenţă. Au fost menţionate trei tipuri de secvenţe – liste, tupluri şi şiruri şi operaţia de **feliere**, care ne permite să extragem o parte (engl. slice) din secvenţă.

Exemplu:

```
#!/usr/bin/python
# Fişier: seq.py

shoplist = ['mere', 'mango', 'morcovi', 'banane']
nume = 'swaroop'

# Operaţia de indexare sau 'subscriere'
print('Itemul 0 este', shoplist[0])
print('Itemul 1 este', shoplist[1])
print('Itemul 2 este', shoplist[2])
print('Itemul 3 este', shoplist[3])
print('Itemul -1 este', shoplist[-1])
print('Itemul -2 este', shoplist[-2])
print('Caracterul 0 este', nume[0])

# Felierea unei liste
print('Itemurile de la 1 la 3 sunt', shoplist[1:3])
print('Itemurile de la 2 la sfârsit sunt', shoplist[2:])
print('Itemurile de la 1 la -1 sunt', shoplist[1:-1])
print('Itemurile de la început la sfârşit sunt', shoplist[:])

# Felierea unui şir
print('Caracterele de la 1 la 3 sunt', nume[1:3])
print('Caracterele de la 2 la end sunt', nume[2:])
print('Caracterele de la 1 la -1 sunt', nume[1:-1])
```

```
print('Caracterele de la început la sfârşit sunt ', nume[:])
```

Rezultat:

```
   $ python seq.py

   Itemul 0 este mere

   Itemul 1 este mango

   Itemul 2 este morcovi

   Itemul 3 este banane

   Itemul -1 este banane

   Itemul -2 este morcovi

   Caracterul 0 este s

   Itemurile de la 1 la 3 sunt ['mango', 'morcovi']

   Itemurile de la 2 la sfârşit sunt ['morcovi', 'banane']

   Itemurile de la 1 la -1 sunt ['mango', 'morcovi']

   Itemurile de la început la sfârşit sunt ['mere', 'mango', 'morcovi',
'banane']

   Caracterele de la 1 la 3 sunt wa

   Caracterele de la 2 la sfârşit sunt aroop

   Caracterele de la 1 la -1 sunt waroo

   Caracterele de la început la sfârşit sunt swaroop
```

Cum funcţionează:

La început, vedem cum se folosesc indecşii pentru a obţine un element anume din secvenţă. Asta se mai numeşte *operaţia de subscriere*. De câte ori specificaţi un număr întro pereche de paranteze drepte alăturate unei secvenţe, Python vă va extrage itemul corespunzator poziţiei în secvenţă. Reţineţi că Python începe numărătoarea de la 0. Astfelshoplist[0] extrage primul item şi shoplist[3] îl extrage pe al patrulea din secvenţa shoplist.

Indexul poate fi şi un număr negativ, caz în care poziţia este calculată de la sfârşitul secvenţei. Din acest motiv shoplist[-1] indică ultimul item din secvenţă, iar shoplist[-2]penultimul item. Operaţia de feliere este folosită specificând numele secvenţei urmat de o pereche opţională de numere separate prin semnul doua puncte (engl. colon) incluse în paranteze drepte. Observaţi că este foarte asemănător cu operaţia de indexare folosită până acum. De reţinut că numerele sunt opţionale, semnul două puncte NU este opţional.

Primul număr (înainte de două puncte) în operaţia de feliere indică punctul unde începe felia, iar al doilea număr indică unde se temină. Dacă nu este specificat primul număr, Python va începe cu începutul secvenţei. Dacă este omis al doilea număr, Python se va opri la sfârşitul secvenţei. Observaţi că felia întoarsă (uneori se spune 'returnată') *începe*cu poziţia dată de primul număr, dar se încheie imediat înainte de poziţia dată de al doilea număr adică începutul este inclus, sfârşitul nu este inclus în felie.

Astfel, shoplist[1:3] întoarce o felie din secvenţa care începe cu poziţia 1, include poziţia 2, dar nu include poziţia 3 deci este întoarsă o felie de două itemuri. Similar, shoplist[:]întoarce o copie a secvenţei.
Mai puteti face feliere şi cu indecşi negativi. Numerele negative sunt folosite pentru a indica poziţii de la sfârşitul secvenţei. De exemplu, shoplist[:-1] va întoarce o felie din secvenţă care exclude ultimul item din secvenţă, dar include tot restul secvenţei.
De asemenea, putem da al treilea argument pentru feliere, care devine *pasul* de feliere (implicit pasul este 1):

```
>>> shoplist = ['mere', 'mango', 'morcovi', 'banane']
>>> shoplist[::1]
['mere', 'mango', 'morcovi', 'banane']
>>> shoplist[::2]
['mere', 'morcovi']
>>> shoplist[::3]
['mere', 'banane']
>>> shoplist[::-1]
['banane', 'morcovi', 'mango', 'mere']
```

Iată ce se întâmplă când pasul este 2, obţinem itemurile cu poziţiile 0, 2, … Dacă pasul este 3, obtinem itemurile din poziţiile 0, 3, etc.

Încercaţi variate combinaţii de specificaţii de feliere în modul interactiv, pentru a vedea imediat rezultatele. Marele avantaj al secvenţelor este că puteţi accesa tuplurile, listele şi şirurile în acelaşi fel!

Seturi

Seturile sunt colecţii *neordonate* de obiecte simple. Acestea sunt folosite atunci cand existenţa unui obiect în colecţie este mai importantă decât poziţia lui sau numărul de apariţii.

Folosind seturi, puteţi testa apartenenţa, dacă un set este subset al altui set, puteţi afla intersecţia a două seturi şi aşa mai departe.

```
>>> bri = set(['brazilia', 'rusia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('rusia')
>>> bri & bric # OR bri.intersection(bric)
{'brazilia', 'india'}
```

Cum funcţionează:

Exemplul este autoexplicativ deoarece implică teoria de bază învăţată la şcoala despre mulţimi (seturi).

Referinţe

Când creaţi un obiect şi îi atribuiţi o valoare, variabila doar indică obiectul creat, nu reprezintă obiectul însuşi! Asta înseamnă că numele variabilei este un indicator către acea parte a memoriei calculatorului în care este stocat obiectul. Acest fapt se numeşte**legare** (engl. binding) a numelui la obiect.

În general, nu trebuie să vă îngrijoraţi de asta, dar există un efect subtil datorat referinţei de care trebuie să fiţi conştienţi:

Exemplu:

```
#!/usr/bin/python
# Fişier: reference.py

print('Atribuire simplă')
lista_iniţială = ['mere', 'mango', 'morcovi', 'banane']
lista_mea = lista_iniţială # lista_mea este doar un alt nume al aceluiaşi obiect!

del lista_iniţială[0] # Am cumpărat primul item, deci să-l ştergem din listă

print('lista initială este', lista_iniţială)
print('lista mea este', lista_mea)
# Observaţi că ambele liste apar fără 'mere', confirmând
```

```
# astfel că ele indică acelaşi obiect

print('Copiere făcând o felie intergală')
lista_mea = lista_iniţială[:] # Face o copie prin feliere integrală
del lista_mea[0] # eliminăm primul item

print('lista_iniţială este', lista_iniţială)
print('lista_mea este', lista_mea)
# Observaţi că acum cele două liste sunt diferite?
```

Rezultat:

```
$ python reference.py

Atribuire simplă

lista_iniţială este ['mango', 'morcovi', 'banane']

lista_mea este ['mango', 'morcovi', 'banane']

Copiere făcând o felie intergală

lista_iniţială este ['mango', 'morcovi', 'banane']

lista_mea este ['morcovi', 'banane']
```

Cum funcţionează:

Partea principală a explicaţiei se regăseşte în comentarii.

Reţineţi că dacă vreţi să faceţi o copie a unei liste sau a unor obiecte complexe (nu simple*obiecte*, cum ar fi întregii), atunci trebuie să folosiţi felierea. Dacă folosiţi atribuirea obţineţi încă un nume pentru acelaşi obiect şi asta poate aduce probleme dacă nu sunteţi atenţi.

> **Notă pentru programatorii în Perl**
>
> Reţineţi că o declaraţie de atribuire pentru liste **nu** creeaza o copie. Va trebui să folosiţi felierea pentru a face o copie a secvenţei.

Alte detalii despre şiruri

Am discutat deja în detaliu despre şiruri. Ce ar mai putea fi de ştiut? Ei bine, ştiaţi că şirurile sunt obiecte şi au metode care fac orice de la verificarea unor părţi ale şirului până la eliminarea spaţiilor?

Sirurile pe care le folosiţi în programe sunt obiecte din clasa str. Câteva metode utile sunt arătate în exemplul următor. Pentru o listă completă a metodelor, citiţi help(str).
Exemplu:

```
#!/usr/bin/python
# Fişier: str_methods.py

nume = 'Swaroop'  # Acesta este obiectul şir

if name.startswith('Swa'):
      print('Da, şirul începe cu "Swa"')

if 'a' in nume:
      print('Da, şirul conţine subşirul "a"')

if name.find('war') != -1:
      print('Da, şirul conţine subşirul "war"')

delimitator = '_*_'
Lista_mea= ['Brazilia', 'Rusia', 'India', 'China']
print(delimitator.join(lista_mea))
```

Rezultat:

```
   $ python str_methods.py

   Da, şirul începe cu "Swa"

   Da, şirul conţine subşirul "a"

   Da, şirul conţine subşirul "war"

   Brazilia_*_Rusia_*_India_*_China
```

Cum funcţionează:

Aici vedem o mulţime de metode în acţiune. Metoda startswith este folosită pentru a testa dacă şirul începe sau nu cu un şir dat. Operatorul in se foloseşte pentru a verifica includerea unui şir în şirul dat.
Metoda find este folosită pentru a găsi poziţia unui şir dat în şirul iniţial, care întoarce -1 dacă nu a găsit nimic. Clasa str are şi o metodă simpatică join pentru a alătura itemurile dintro secvenţă, cu un şir pe post de delimitator între itemuri şi intoarce un şir lung generat din toate acestea.

Rezumat

Am explorat diversele structuri de date predefinite în Python în detaliu. Aceste structuri de date vor deveni esenţiale pentru scrierea de programe de dimensiuni rezonabile.

Acum că avem o mulţime de elemente Python asimilate, vom vedea cum se proiectează şi se scriu programe Python în lumea de zi cu zi.

Python ro:Rezolvarea problemelor

Am explorat diverse părţi din limbajul Python şi acum vom vedea cum conlucrează acestea prin proiectarea şi scrierea unui program care *face* ceva util. Ideea este de a învăţa cum să scriem un program Python propriu.

Contents

Problema

Problema este *"Vreau un program care să facă un backup al tuturor fişierelor mele importante"*.

Deşi aceasta este o problema simplă, nu avem destule informaţii pentru a începe găsirea unei soluţii. Se impune o **analiză** suplimentară. De exemplu, cum specificăm *care* fişiere trebuie salvate? *Cum* vor fi ele stocate? *Unde* vor fi stocate?

După o analiză corectă a problemei, *proiectăm* programul. Facem o listă cu lucruri care descriu cum trebuie să funcţioneze programul nostru. În acest caz, am creat lista următoare care descrie cum vreau *EU* să meargă. Dacă faceţi voi proiectarea s-ar putea să rezulte o altfel de analiză, întrucât fiecare face lucrurile în felul lui, deci e perfect OK.

1. Fişierele şi directoarele care trebuie salvate sunt specificate într-o listă.
2. Backup-ul trebuie stocat în directorul principal de backup
3. Fişierele sunt stocate întro arhivă zip.
4. Numele arhivei zip este data şi ora.
5. Folosind comanda standard zip disponibilă implicit în toate distribuţiile Linux/Unix. Utilizatorii de Windows pot instala din pagina proiectului GnuWin32 şi adaugaC:\Program Files\GnuWin32\bin la variabila de mediu PATH, similar modului în caream făcut pentru recunoaşterea însăşi a comenzii python. Reţineţi că puteţi folosi orice comandă de arhivare

atât timp cât această are o interfaţă linie de comandă, ca să îi putem transmite argumente din programul nostru.

Soluţia

Întrucât designul programului nostru este relativ stabil, putem scrie codul care*implementează* soluţia noastră.

```
#!/usr/bin/python
# Fişier: backup_ver1.py

import os
import time

# 1. Fişierele şi directoarele de salvat sunt specificate într-o listă.
source = ['"C:\\My Documents"', 'C:\\Code']
# Observaţi că a fost nevoie de ghilimele duble în interiorul şirului pentru a proteja spaţiile din interiorul numelor.

# 2. Salvarea (engl. backup) trebuie stocată în directorul principal de backup
target_dir = 'E:\\Backup'  # Nu uitaţi să schimbaţi asta cu directorul folosit de voi

# 3. Fişierele sunt salvate întro arhivă zip.

# 4. Numele arhivei zip este data şi ora curentă
target = target_dir + os.sep + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. Folosim comanda zip pentru a include fişierele şi directoarele de salvat în arhivă
zip_command = "zip -qr {0} {1}".format(target, ''.join(source))

# Rulăm comanda de backup
if os.system(zip_command) == 0:
    print('Salvare reuşită în ', target)
else:
    print('Backup EŞUAT')
```

Rezultat:

```
$ python backup_ver1.py

Salvare reuşită în E:\Backup\20090208153040.zip
```

Acum suntem în faza de **testare** în care verificăm dacă programul nostru lucrează corect. Dacă nu se comportă cum trebuie, va fi nevoie de **debugging** adică eliminarea erorilor din program.

Dacă programul nu va merge, puneţi o declaraţie print(zip_command) imediat înainte de apelul os.system şi rulaţi programul. Acum copiaţi comanda zip_command la promptul shell-ului şi verificaţi dacă merge pe cont propriu. Dacă aceasta eşuează, citiţi manualul comenzii zip ca să aflaţi ce ar putea fi greşit. Dacă reuşeşte, verificaţi programul Python ca să vedeţi dacă este exact ca mai sus.

Cum funcţionează:

Veţi observa cum am transformat *designul* în *cod* întro manieră pas cu pas.

Utilizăm modulele os şi time importându-le de la început. Apoi specificăm directoarele care trebuie salvate în lista source. Directorul destinaţie este locul unde stocăm toate salvările şi acesta este specificat în variabila target_dir. Numele arhivei zip pe care o vom crea este "data curentă+ora curentă" pe care le găsim folosind funcţia time.strftime(). Arhiva va avea extensia .zip şi va fi stocată în directorul target_dir.
Observaţi folosirea variabilei os.sep – cea care ne dă separatorul de director al sistemului vostru de operare; acesta va fi '/' în Linux şi Unix, '\\' în Windows şi ':' în Mac OS. Folosirea declaraţiei os.sep în locul acestor caractere va face programul mai portabil între aceste sisteme.
Funcţia time.strftime() primeşte o specificaţie ca aceea folosită în program. Specificaţia%Y va fi înlocuită cu anul, fără secol. Specificaţia %m va fi înlocuită cu luna, ca numar zecimal între 01 şi 12 ş.a.m.d. Lista completă a specificaţiilor poate fi găsită în Manualul de referinţă Python.
Creăm numele directorului destinaţie folosind operatorul de adunare care *concatenează*şirurile adică alătură şirurile şi produce unul mai lung. Atunci noi creăm un şirzip_command care conţine comanda pe care o vom executa. Puteţi verifica dacă a rezultat o comandă corectă prin executarea ei de sine stătătoare într-un shell (terminal Linux sau DOS prompt).
Comanda zip pe care o folosim are câteva opţiuni şi parametri transmişi. Opţiunea -q este folosită pentru a indica modul de lucru *tăcut* (engl. **q**uiet). Opţiunea -r specifică modul recursiv de parcurgere a directoarelor, adică trebuie să includă şi toate subdirectoarele şi subdirectoarele acestora etc. Cele două opţiuni se combină şi se specifică pe scurt -qr. Opţiunile sunt urmate de numele arhivei care va fi creată urmată de lista fişierelor şi directoarelor de salvat. Convertim lista source într-un şir folosind metoda join a şirurilor, pe care am învăţat deja s-o folosim.
În fine, *rulăm* comanda folosind funcţia os.system care execută comanda ca şi cum ar fi fost lansată din *sistem* adică în shell – ea întoarce 0 dacă comanda a fost executată cu succes, altfel întoarce un cod de eroare.
În funcţie de rezultatul comenzii, tipărim pe ecran mesajul adecvat, cum că salvarea a reuşit sau nu.

Asta e, am creat un script care să faca un backup al fişierelor importante din sistem!

Notă pentru utilizatorii de Windows

> În locul secvenţelor de evadare cu dublu backslash, puteţi folosi şi şiruri brute. De exemplu, folosiţi 'C:\\Documents' sau r'C:\Documents'. În orice caz, **nu** folosiţi'C:\Documents' întrucât veţi ajunge să folosiţi o secvenţă de evadare necunoscută,\D.

Acum că avem un script funcţional de salvare, îl putem rula de câte ori vrem să obţinem o salvare a fişierelor. Utilizatorii de Linux/Unix sunt sfătuiţi să folosească metode executabileaşa cum am discutat în capitolele precedente, astfel ca ele să poată rula de oriunde, oricând. Asta se numeşte faza de **operare** sau de **distribuire** a software-ului.
Programul de mai sus funcţionează corect, dar (de obicei) primul program nu funcţionează cum ne-am aştepta. De exemplu ar putea fi probleme dacă nu am proiectat corect programul sau dacă avem o eroare de dactilografiere în scrierea codului (engl. typo), etc. În modul adecvat, vă veţi întoarce la faza de design sau debuggigg pentru a rezolva problema.

A doua versiune
Prima versiune a scriptului nostru funcţionează. Totuşi, putem rafina programul pentru a lucra mai bine în utilizarea sa de zi cu zi. Asta se numeşte **întreţinere** sau mentenanţă a software-ului (engl. maintenance).

Unul din rafinamentele pe care le-am considerat eu utile a fost un mecanism mai bun de denumire a salvărilor, folosind *ora* ca nume al fişierului, iar data ca nume de subdirector al directorului de backup care să conţină salvările din aceeaşi *data*. Primul avantaj este că salvările vor fi stocate într-o manieră ierarhică şi vor fi mai uşor de gestionat. Al doilea avantaj este că lungimea numelor de fişier va fi mai mult mai mică. Al treilea avantaj este că se va putea verifica mai uşor dacă au fost făcute salvări zilnic (în ziua în care nu s-a facut, directorul având ca nume acea dată lipseşte, întrucât nu a fost creat).

```
#!/usr/bin/python
# Fişier: backup_ver2.py

import os
import time

# 1. Fişierele şi directoarele de salvat sunt specificate întro listă.
source = ['"C:\\My Documents"', 'C:\\Code']
# Observaţi că au fost necesare ghilimele duble pentru a proteja spaţiile din interiorul numelor.

# 2. Salvarea trebuie stocată în directorul principal de backup
target_dir = 'E:\\Backup'  # Nu uitaţi să schimbaţi asta cu directorul pe care îl folosiţi voi

# 3. Fişierele sunt salvate în fişiere zip.
# 4. Data curentă este numele subdirectorului din folderul principal
azi = target_dir + os.sep + time.strftime('%Y%m%d')
# Ora curentă este numele arhivei zip
acum = time.strftime('%H%M%S')

# Creăm subdirectorul, dacă nu exista înainte
if not os.path.exists(azi):
    os.mkdir(azi)  # creăm directorul
    print('Am creat cu succes directorul ', azi)

# Numele fişierului arhiva zip
```

```
target = azi + os.sep + acum + '.zip'

# 5. Folosim comanda zip pentru a colecta fişierele în arhivă.
zip_command = "zip -qr {0} {1}".format(target, ''.join(source))

# Rulăm programul de salvare
if os.system(zip_command) == 0:
    print('Salvare reuşită în ', target)
else:
    print('Salvare EŞUATĂ')
```

Rezultat:

```
$ python backup_ver2.py

Am creat cu succes directorul E:\Backup\20090209

Salvare reuşită în E:\Backup\20090209\111423.zip



$ python backup_ver2.py

Salvare reuşită în E:\Backup\20090209\111837.zip
```

Cum funcţionează:

Majoritatea codului ramâne neschimbat. Schimbările constau un testarea existenţei directorului având ca nume data curentă în interiorului directorului principal de backup folosind funcţia os.path.exists. Dacă acesta nu există, îl creăm noi folosind funcţiaos.mkdir.

Versiunea a treia

A doua versiune merge bine când facem multe salvări, dar e greu de văzut ce este salvat în fiecare arhiva! De exemplu, poate am făcut o schimbare mare unui program sau unei prezentări şi aş vrea să asociez aceste schimbări cu numele programului sau prezentării şi arhiva zip. Aceasta se poate realiza uşor prin ataşarea unui comentariu furnizat de utilizator la numele arhivei zip.

Notă

Următorul program nu funcţionează, deci nu va alarmaţi, urmaţi-l totuşi, pentru că e o lecţie în el.

```
#!/usr/bin/python
# Fişier: backup_ver3.py

import os
import time
```

```
# 1. Fişierele şi directoarele de salvat sunt specificate întro listă.
source = ['"C:\\My Documents"', 'C:\\Code']
# Observaţi că a fost nevoie de ghilimele duble pentru a proteja spaţiile din interiorul numelor.

# 2. Salvarea trebuie stocată în directorul principal
target_dir = 'E:\\Backup'  # Nu uitaţi să schimbaţi asta cu ce folosiţi voi
# 3. Fişierele sunt salvate întro arhivă zip.
# 4. Data curentă este numele subdirectorului
azi = target_dir + os.sep + time.strftime('%Y%m%d')
# Ora curentă este numele arhivei zip
acum = time.strftime('%H%M%S')

# Luăm un comentariu de la utilizator pentru a crea numele
comentariu = input('Introduceţi un comentariu --> ')
if len(comentariu) == 0: # Verificaţi dacă a fost introdus
    target = azi + os.sep + acum+ '_' +
    comentariu.replace(' ', '_') + '.zip'

# Creăm subdirectorul dacă nu există deja
if not os.path.exists(azi):
    os.mkdir(azi)  # Creăm directorul
    print('Am creat cu succes directorul ', azi)

# 5. Folosim comanda zip pentru a colecta fişierele în arhiva
zip_command = "zip -qr {0} {1}".format(target, ''.join(source))

# Rulăm salvarea
if os.system(zip_command) == 0:
    print('Salvare reuşită în ', target)
else:
    print('Salvare EŞUATĂ')
```

Rezultat:

```
$ python backup_ver3.py

  File "backup_ver3.py", line 25

    target = azi + os.sep + now + '_' +

                                        ^


SyntaxError: invalid syntax
```

Cum (nu) funcţionează:

Acest program nu funcţionează! Python spune că e undeva o eroare de sintaxă ceea ce înseamnă că programul nu a fost bine scris, că nu respectă structura pe care se aşteaptă Python să o găsească acolo. Când vedem eroarea dată de Python, aflăm şi locul unde a detectat el eroarea. Deci începem eliminarea erorilor (engl. debugging) programului de la acea linie.

La o observaţie atentă, vedem ca o linie logică a fost extinsă pe două linii fizice fără să se specifice acest lucru. În esenţă Python a găsit operatorul de adunare (+) fără vreun operand în acea linie şi prin urmare nu ştie cum să continue. Vă amintiţi că putem specifica trecerea unei linii logice pe următoarea linie fizică folosind un backslash la sfârşitul liniei fizice. Astfel facem corectura la progrmul nostru. Aceasta corecţie a programului când gasim erori se numeşte **depanare** (engl. bug fixing).

Versiunea a patra

```
#!/usr/bin/python
# Fişier: backup_ver4.py

import os
import time

# 1. Fişierele de salvat sunt specificate întro listă.
source = ['"C:\\My Documents"', 'C:\\Code']
# Observaţi că a trebuit să punem ghilimele duble pentru a proteja spaţiile din interiorul numelor.

# 2. Salvarea trebuie stocată în directorul principal de backup
target_dir = 'E:\\Backup'  # Nu uiţati să schimbaţi asta cu ceea ce folosiţi voi

# 3. Salvările se fac în arhive zip.

# 4. Ziua curentă este numele subdirectorului din directorul principal de backup
azi = target_dir + os.sep + time.strftime('%Y%m%d')
# Ora curentă este numele arhivei zip
acum = time.strftime('%H%M%S')

# Acceptăm un comentariu de la utilizator
comentariu = input('Introduceţi un comentariu --> ')
if len(comentariu) == 0: # Verificăm dacă a fost introdus un comentariu
    target = azi + os.sep + acum + '.zip'
else:
    target = azi + os.sep + acum + '_' + \
        comentariu.replace(' ', '_') + '.zip'

# Creăm subdirectorul, dacă nu exista deja
if not os.path.exists(azi):
    os.mkdir(azi)  # creăm directorul
    print('Am creat cu succes directorul ', today)

# 5. Folosim comanda zip pentru a colecta fişierele în arhivă zip
zip_command = "zip -qr {0} {1}".format(target, ''.join(source))

# Rulăm comanda de backup
if os.system(zip_command) == 0:
    print('Salvare reuşită în ', target)
else:
    print('Salvare EŞUATĂ')
```

Rezultat:

```
   $ python backup_ver4.py


   Introduceţi un comentariu --> noi exemple adăugate
```

```
Salvare reuşită în E:\Backup\20090209\162836_noi_exemple_adăugate.zip



$ python backup_ver4.py

Introduceţi un comentariu -->

Salvare reuşită în E:\Backup\20090209\162916.zip
```

Cum functionează:

Acest program funcţionează, acum! Să parcurgem îmbunătăţirile pe care i le-am adus în versiunea 3. Acceptăm un comentariu de la utilizator folosind funcţia input şi apoi testăm dacă s-a introdus ceva sau nu calculând lungimea şirului introdus cu ajutorul funcţiei len. Dacă utilizatorul a dat ENTER fără să introducă ceva (poate era doar un backup de rutină şi n-au fost făcute modificări anume), apoi continuăm ca mai înainte.
Totuşi, dacă a fost introdus un comentariu, acesta este ataşat numelui arhivei zip, chiar înaintea extensiei .zip. Observaţi că înlocuim spaţiile în comentariu cu underscore – ca să fie mai uşoară gestiunea fişierelor cu nume lungi.

Alte rafinamente

A patra versiune este una satisfăcătoare pentru majoritatea utilizatorilor, dar este mereu loc pentru mai bine. De exemplu se poate introduce un nivel de *logoree* (engl. verbosity) pentru program, cu ajutorul opţiunii -v Pentru a face programul mai vorbăreţ.
Altă îmbunătăţire posibilă ar fi să permitem ca fişierele şi directoarele de salvat să fie transmise scriptului la linia de comandă. Noi le putem culege din lista sys.argv şi le putem adăuga la variabila listă source folosind metoda extend a clasei list.
Cea mai importanta extindere ar fi să nu folosim os.system ci direct modulele predefinitezipfile sau tarfile pentru a crea aceste arhive. Ele sunt parte a bibliotecii standard şi sunt deja disponibile pentru a scrie un program fără dependente externe pentru programul de arhivare. Totuşi, am folosit os.system pentru crearea salvărilor din motive pedagogice, pentru ca exemplul să fie destul de simplu de înteles, dar şi util.
Puteţi încerca să scrieţi a cincea variantă folosind modulul zipfile în locul apeluluios.system?

Procesul de dezvoltare de software

Am trecut prin diverse **faze** în procesul de scriere a unui program. Aceste faze pot fi rezumate astfel:

1. Ce (Analiza)
2. Cum (Design)
3. Executare (Implementare)

4. Test (Testare şi eliminare erori)
5. Utilizare (Operare sau distribuire)
6. Mentenanţă (Rafinare)

O cale recomandată de a scrie programe este procedura urmată de noi la scrierea scriptului de salvare: facem analiza şi designul. Începem implementarea cu o versiune simplă. O testăm şi o depanăm. O folosim pentru a ne asigura că merge cum ne-am propus. Acum îi adăugăm noi facilităţi pe care le dorim şi parcurgem ciclul FACI – TESTEZI – UTILIZEZI de câte ori este nevoie. Reţineţi, **Software-ul este crescut, nu construit.**

Rezumat

Am văzut cum se creează un program/script Python şi diferite stadii implicate de scrierea unui program. Aţi putea considera utilă scrierea de programe proprii, atât pentru acomodarea cu Python cât şi pentru a rezolva probleme.

În continuare vom discuta despre programarea orientată pe obiecte.

Python ro:Programare orientată pe obiecte

Contents

Introducere

În toate programele folosite până acum ne-am construit soluţiile în jurul funcţiilor adică blocuri de declaraţii care manipulează date. Acest mod de programare se numeşte*orientat pe proceduri*. Există şi un alt mod de organizare a programelor, în care funcţionalitatea şi datele sunt împachetate împreună în unităţi numite obiecte. Acest mod de structurare defineşte paradigma "orientat pe obiecte. *Aproape tot timpul puteţi folosi abordări procedurale în programare, dar când scrieţi programe mari sau aveţi de rezolvat probleme care sunt mai aproape de acest mod de structurare, puteţi folosi tehnicile de programare orientată pe obiecte.*.

Clasele şi obiectele sunt două aspecte ale programării orientate pe obiecte. O **clasă**creeaza un nou *tip* în care *obiectele* sunt *instanţe* ale clasei. O analogie este că puteţi avea variabile de tip int care se traduce prin aceea că variabilele care stochează întregi sunt instanţe (obiecte) ale clasei int.

Notă pentru programatorii în limbaje cu tipuri statice

Observaţi că până şi întregii sunt trataţi ca obiecte (ale clasei int). Asta e diferit de C++ şi Java (în versiunile dinainte de 1.5) în care întregii sunt tipuri primitive native. A se vedea help(int) pentru detalii despre clasă.

Programatorii în C# şi Java 1.5 vor găsi o similaritate cu conceptele de încapsulare şi decapsulare.

Obiectele pot stoca date folosind variabile obişnuite care *aparţin* obiectului. Variabilele care aparţin unui obiect sunt numite **câmpuri**. Obiectele pot avea şi funcţionalitate folosind funcţii care *aparţin* clasei. Aceste funcţii se munesc **metode** ale clasei. Această terminologie este importantă deoarece ne ajuta să diferenţiem între funcţii şi variabile independente şi cele aparţinând unui obiect sau unei clase. Împreună, variabilele şi funcţiile care aparţin unei clase se numesc **atribute** ale clasei.

Câmpurile sunt de două tipuri – ele pot aparţine fiecărei instanţe/obiect al clasei sau pot aparţine însăşi clasei. Acestea sunt numite **variabile de instanţă** respectiv **variabile ale clasei**.

O clasă este creată folosind cuvântul cheie class. Câmpurile şi metodele clasei sunt listate întrun bloc indentat.

self

Clasele şi metodele au o diferenţă specifică faţă de funcţiile obişnuite – ele trebuie să aibă un *prenume* suplimentar care trebuie adăugat la începutul listei de parametri, dar **nu trebuie** să-i daţi o valoare când apelaţi metoda, Python o va furniza. Această variabilă specială se referă la obiectul *însuşi* (engl. self) şi prin convenţie este numită self.

Cu toate acestea, deşi puteţi să-i daţi orice nume acestui parametru, este *puternic recomandat* să folosiţi numele self – orice alt nume este dezaprobat. Există multe avantaje în folosire numelui standard – orice cititor al programului va înţelege imediat despre ce este vorba şi chiar mediile IDE (Integrated Development Environments) specializate te pot ajuta dacă foloseşti self.

Notă pentru programatorii în C++/Java/C#

self din Python este echivalent cu pointerul this din C++ şi cu referinţa this din Java şi C#.

Probabil vă întrebaţi cum dă Python valoarea corectă lui self şi de ce nu trebuie să-i dăm o valoare. Un exemplu va clarifica această problemă. Să zicem că aveţi o clasă numităMyClass şi o instanţă a acestei clase, numită myobject. Când apelaţi o metodă a acestui obiect myobject.method(arg1, arg2), apelul este automat convertit de Python înMyClass.method(myobject, arg1, arg2) – asta e toată marea specialitate a lui self.
Asta înseamnă şi că dacă aveţi o metodă care nu primeşte argumente, tot va trebui să aveţi un argument – self.

Clase

Cea mai simplă clasă posibilă este arătată în exemplul următor.

```
#!/usr/bin/python
# Fişier: simplestclass.py

class Persoana:
    pass  # Un bloc gol

p = Persoana()
print(p)
```

Rezultat:

```
$ python simplestclass.py

<__main__.Persoana object at 0x019F85F0>
```

Cum funcţionează:

Creăm o clasă nouă folosind declaraţia class şi numele clasei. Aceasta este urmată de un bloc indentat de declaraţii care formează corpul clasei. În acest caz avem un bloc gol, arătat de declaraţia pass.
În continuare creăm un obiect/instanţă a acestei clase folosind numele clasei urmat de o pereche de paranteze. (Vom învăţa mai multe despre instanţiere în paragraful următor). Pentru propria verificare, confirmăm tipul variabilei prin simpla ei tipărire. Aflăm că avem o instanţă a variabilei din clasa Persoana din modulul __main__.
Observaţi că a fost tipărită şi adresa unde este stocat obiectul în memoria calculatorului. Adresa aceasta va avea o valoare diferită în alt calculator deoarece Python îl stochează unde are loc.

Metodele obiectelor

Am discutat deja că obiectele/clasele pot avea metode, exact ca funcţiile, doar că au un argument self în plus. Iată un exemplu.

```
#!/usr/bin/python
# Fişier: metoda.py

class Persoana:
```

```
    def ziSalut(self):
        print('Salut, ce mai faci?')

p = Persoana()
p.ziSalut()

# Acest exemplu poate fi scris şi ca Persoana().ziSalut()
```

Rezultat:

```
$ python metoda.py

Salut, ce mai faci?
```

Cum funcţionează:

Aici vedem particula self în acţiune. Observaţi că metoda ziSalut nu primeşte parametri, dar tot are argumentul self în definiţia funcţiei.

Metoda __init__

Există multe nume de metode care au un înteles special în clasele Python. Acum vom afla semnificaţia metodei __init__.

Metoda __init__ este executată imediat ce este instanţiat un obiect al clasei. Metoda este utilă pentru a face *iniţializarea* dorită pentru obiectul respectiv. Observaţi că numele este încadrat cu dublu underscore.

Exemplu:

```
#!/usr/bin/python
# Fişier: class_init.py

class Persoana:
    def __init__(self, nume):
        self.nume = nume
    def ziSalut(self):
        print('Salut, numele meu este', self.nume)

p = Persoana('Swaroop')
p.ziSalut()

# Acest exemplu putea fi scris Persoana('Swaroop').ziSalut()
```

Rezultat:

```
$ python class_init.py

Salut, numele meu este Swaroop
```

Cum funcţionează:

Definim metoda __init__ să ia un parametru nume (pe lângă obişnuitul self). În acest caz creăm doar un câmp numit nume. Observaţi că deşi ambele sunt numite ‘nume’, cele două sunt obiecte diferite. Notaţia cu punct ne permite să le deosebim.
Cel mai important, observaţi că nu apelăm explicit metoda __init__ ci îi transmitem argumentele în paranteză după numele clasei în momentul creării obiectului/instanţa a clasei. Aceasta este semnificaţia acestei metode.
Acum putem să folosim câmpul self.name în metodele noastre, ceea ce este arătat în metoda ziSalut.

Variabile de clasă, variabile de instanţă

Am discutat deja partea de funcţionalitate a claselor şi obiectelor (adică metodele), acum să învăţăm ceva despre partea de date. Partea de date, aşa-numitele câmpuri, nu sunt altceva decât variabile obişnuite care sunt *legate* de **spaţiile de nume** ale claselor şi obiectelor. Asta înseamnă că aceste nume sunt valabile numai în contextul claselor şi obiectelor respective. Din acest motiv acestea sunt numite *spaţii de nume* (engl. name spaces).

Există două feluri de *câmpuri* – variabile de clasa şi variabile de obiect/instanţă, care sunt clasificate în funcţie de *proprietarul* variabilei.

Variabilele de clasă sunt partajate – ele pot fi accesate de toate instanţele acelei clase. Există doar un exemplar al variabilei de clasă şi când o instanţă îi modifică valoarea, această modificare este văzută imediat de celelalte instanţe.

Variabilele de instanţă sunt proprietatea fiecărei instanţe a clasei. În acest caz, fiecare obiect are propriul exemplar al acelui câmp adică ele nu sunt relaţionate în nici un fel cu câmpurile având acelaşi nume în alte insţante. Un exemplu va ajuta la înţelegerea situaţiei:

```
#!/usr/bin/python
# Fişier: objvar.py

clasa Robot:
    '''Reprezintă un robot cu nume.'''

    # O variabilă de clasă, numărătorul populaţiei de roboţi
    populaţie = 0

    def __init__(self, nume):
        '''Iniţializează datele.'''
        self.nume = nume
        print('(Iniţializez robotul {0})'.format(self.nume))

        # Când această instanţă este creată, robotul se
        # adaugă la populaţie
        Robot.populaţie += 1

    def __del__(self):
```

```
        '''Dispar...'''
        print('{0} este dezmembrat!'.format(self.nume))

        Robot.populaţie -= 1

        if Robot.populaţie == 0:
            print('{0} a fost ultimul.'.format(self.nume))
        else:
            print('Mai există {0:d} roboţi apţi de lucru.'.format(Robot.populaţie))

    def ziSalut(self):
        '''Salutare de la robot.

    Da, pot să facă şi asta.'''
        print('Salut. Stăpânii mei îmi zic {0}.'.format(self.nume))

    def câţi():
        '''Tipăreşte populaţia curentă.'''
        print('Avem {0:d} roboţi.'.format(Robot.populaţie))
    câţi = staticmethod(câţi)

droid1 = Robot('R2-D2')
droid1.ziSalut()
Robot.câţi()

droid2 = Robot('C-3PO')
droid2.ziSalut()
Robot.câţi()

print("\nRoboţii pot să facă nişte treabă aici.\n")

print("Roboţii au terminat treaba. Deci să-i distrugem.")
del droid1
del droid2

Robot.câţi()
```

Rezultat:

```
(Iniţializez robotul R2-D2)

Salut. Stăpânii mei îmi zic R2-D2.

Avem 1 roboţi.

(Iniţializez robotul  C-3PO)

Salut. Stăpânii mei îmi zic C-3PO.

Avem 2 roboţi.



Roboţii pot să facă nişte treabă aici.
```

```
Roboţii au terminat treaba. Deci să-i distrugem.

R2-D2 este dezmembrat!

Mai există 1 roboţi apţi de lucru.

C-3PO este dezmembrat!

C-3PO a fost ultimul.

Avem 0 roboţi.
```

Cum funcţionează:

Este un exemplu lung, dar ajută la evidenţierea naturii variabilelor de clasă şi de instanţă. Aici câmpul populaţie aparţine clasei Robot şi este deci o variabilă de clasă. Variabilanume aparţine obiectului (îi este atribuită folosind self.) deci este o variabilă de obiect/instanţă.
Asadar ne referim la variabila de clasă populaţie cu notaţia Robot.populaţie şi nu cuself.populaţie. Ne referim la variabila de instanţă nume cu notaţia self.nume în metodele acelui obiect. Amintiţi-vă această diferenţă simplă între variabilele de clasă şi de instanţă. Mai observaţi şi că o variabilă de obiect cu acelaşi nume ca o variabilă de clasă, va ascunde variabila de clasă faţă de metodele clasei!
Metoda câţi este în fapt o metodă a clasei şi nu a instanţei. Asta înseamnă că trebuie să o definim cu declaraţia classmethod sau staticmethod Dacă vrem să ştim cărui spaţiu de nume îi aparţine. Întrucât nu vrem aceasta informaţie, o vom defini cu staticmethod.
Am fi putut obţine acelaşi lucru folosind decoratori:

```
    @staticmethod
    def câţi():
        '''Tipăreşte populaţia curentă.'''
        print('Avem {0:d} roboti.'.format(Robot.populaţie))
```

Decoratorii pot fi concepuţi ca scurtături pentru apelarea unor declaraţii explicite, aşa cum am văzut în acest exemplu.

Observaţi că metoda __init__ este folosită pentru a iniţializa cu un nume instanţa claseiRobot. În această metodă, mărim populaţie cu 1 intrucât a fost creat încă un robot. Mai observaţi şi că valoarea self.nume este specifică fiecărui obiect, ceea ce indică natura de variabilă de instanţă a variabilei.
Reţineţi că trebuie să vă referiţi la variabilele şi metodele aceluiaşi obiect **numai** cu self. Acest mod de indicare se numeşte *referinţă la atribut*.

În acest program mai vedem şi **docstrings** pentru clase şi metode. Putem accesa docstringul clasei în runtime (rom. timpul execuţiei) folosind notaţia Robot.__doc__ şi docstring-ul metodei ziSalut cu notaţia Robot.ziSalut.__doc__
Exact ca şi metoda __init__, mai există o metodă specială, __del__, care este apelată atunci când un obiect trebuie distrus, adică nu va mai fi folosit, iar resursele lui sunt returnate sistemului. În această metodă reducem şi numărul Robot.populaţie cu 1.
Metoda __del__ este executată dacă obiectul nu mai este în folosinţă şi nu există o garanţie că metoda va mai fi rulată. Dacă vreţi să o vedeţi explicit în acţiune, va trebui să folosiţi declaraţia del cum am făcut noi aici.

Notă pentru programatorii în C++/Java/C#

Toţi membrii unei clase (inclusiv membrii date) sunt *publici* şi toate metodele sunt*virtual* în Python.

O excepţie: Dacă folosiţi membrii date cu nume care *încep cu dublu underscore*precum __var_privată, Python exploatează acest aspect şi chiar va face variabila să fie privată.
Aşadar, convenţia este că orice variabilă care vrem să fie folosită numai în contextul clasei sau obiectului, ar trebui să fie numită cu primul caracter underscore, iar toate celelalte nume sunt publice şi pot fi folosite de alte clase/instanţe. Reţineţi că aceasta este doar o convenţie şi nu este impusă de Python (exceptând prefixul dublu underscore).

Moştenire

Un beneficiu major al programării orientate pe obiecte este *refolosirea* codului şi o cale de a obţine asta este mecanismul de *moştenire*. Moştenirea poate fi descrisă cel mai bine ca şi cum ar implementa o relaţie între un tip şi un subtip între clase.

Să zicem că scrieţi un program în care trebuie să ţineţi evidenţa profesorilor şi studenţilor într-un colegiu. Ei au unele caracteristici comune, precum nume, adresă, vârstă. Ei au şi caracteristici specifice, cum ar fi salariul, cursurile şi perioadele de absenţă, pentru profesori, respectiv notele şi taxele pentru studenţi.

Puteţi crea două clase independente pentru fiecare tip şi procesa aceste clase prin adăugarea de caracteristici noi. Aşa programul devine repede un hăţis necontrolabil.

O cale mai bună ar fi să creaţi o clasă comună numită MembruAlŞcolii şi să faceţi clasele student şi profesor să *moştenească* de la această clasă, devenind astfel subclase ale acesteia, şi apoi să adăugaţi caracteristici la aceste subtipuri.
Această abordare are multe avantaje. Dacă facem vreo schimbare la funcţionalitatea clasei MembruAlŞcolii, ea este automat reflectată şi în subtipuri. De exemplu, puteţi adăuga un nou câmp card ID şi pentru studenţi şi pentru profesori prin simpla adăugare a acestuia la clasa MembruAlŞcolii. Totuşi, schimbările din subtipuri nu afectează alte subtipuri. Alt avantaj este că puteţi face referire la un profesor sau student ca obiectMembruAlŞcolii object, ceea ce poate fi util în anumite situaţii precum calculul numărului de membri ai şcolii. Acest comportament este numit **polimorfism**, în care un subtip poate fi folosit în orice situatie în care se aşteaptă folosirea unui tip părinte adică obiectul poate fi tratat drept instanţă a tipului părinte.
Observaţi şi că *refolosim* codul clasei părinte şi nu e nevoie să-l repetăm în subclase cum am fi fost nevoiţi dacă am fi creat clase independente.

Clasa MembruAlŞcolii în această situaţie este numită *clasa bază* sau *superclasa*.
Claseleprofesor şi student sunt numite *clase derivate* sau *subclase*.
Vom vedea acest exemplu pe un program.

```
#!/usr/bin/python
# Fişier: inherit.py

class MembruAlŞcolii:
     '''Reprezintă orice membru al şcolii.'''
     def __init__(self, nume, vârstă):
          self.nume = nume
          self.varsta = vârstă
          print('(Iniţializez MembruAlŞcolii: {0})'.format(self.nume))

     def descrie(self):
          '''Afişează detaliile mele.'''
          print('Nume:"{0}" Vârstă:"{1}"'.format(self.nume, self.vârstă), end=" ")

class profesor(MembruAlŞcolii):
     '''Reprezintă un profesor.'''
     def __init__(self, nume, vârstă, salariu):
          MembruAlŞcolii.__init__(self, nume, vârstă)
          self.salariu = salariu
          print('(Iniţializez Profesor: {0})'.format(self.nume))

     def descrie(self):
          MembruAlŞcolii.descrie(self)
          print('Salariu: "{0:d}"'.format(self.salariu))

class student(MembruAlŞcolii):
     '''Reprezintă un student.'''
     def __init__(self, nume, vârstă, note):
          MembruAlŞcolii.__init__(self, nume, vârstă)
          self.note = note
          print('(Iniţializez student: {0})'.format(self.nume))
```

```
    def descrie(self):
        MembruAlŞcolii.descrie(self)
        print('Note: "{0:d}"'.format(self.note))

p = profesor('D-na. Shrividya', 40, 30000)
s = student('Swaroop', 25, 75)

print()  # Afişează o linie goală

membri = [p, s]
for membru in membri:
    membru.descrie()  # Funcţionează şi pentru profesor şi pentru student
```

Rezultat:

```
$ python inherit.py

(Iniţializez MembruAlŞcolii: Mrs. Shrividya)

(Iniţializez profesor: D-na. Shrividya)

(Iniţializez MembruAlŞcolii: Swaroop)

(Iniţializez student: Swaroop)



Nume:"D-na. Shrividya" Vârstă:"40" Salariu: "30000"

Nume:"Swaroop" Vârstă:"25" Note: "75"
```

Cum funcţionează:

Pentru a folosi moştenirea, specificăm clasele bază întrun tuplu care urmează numele în definiţia clasei. Apoi observăm că metoda __init__ a clasei bază este apelată explicit folosind variabila self ca să putem iniţializa partea din obiect care provine din clasa bază. Este foarte important de reţinut – Python nu apeleaza automat constructorul clasei bază, trebuie să faceţi asta explicit.

Mai observăm că apelurile către clasa bază se fac prefixind numele clasei apelului metodelor şi punând variabila self împreună cu celelalte argumente.

Se confirmă că folosim instanţele claselor profesor şi student ca şi cum ar fi instanţe ale clasei MembruAlŞcolii când folosim metoda descrie a clasei MembruAlŞcolii.

În plus, observaţi că este apelată metoda descrie a subtipului nu metoda descrie a claseiMembruAlŞcolii. O cale de a întelege asta este că Python începe *întotdeauna* căutarea

metodelor în tipul curent, ceea ce face în acest caz. Dacă nu ar fi găsit metoda, ar fi căutat în clasele părinte, una câte una, în ordinea specificată în tuplul din definiţia clasei.
O notă asupra terminologiei – daca în tuplul de moştenire este listată mai mult de o clasă, acest caz este de *moştenire multiplă*.

Rezumat

Am explorat diverse aspecte ale claselor şi obiectelor precum şi diferite terminologii asociate cu acestea. Am văzut de asemenea beneficiile şi punctele slabe ale programării orientate pe obiecte. Python este puternic orientat pe obiecte şi înţelegerea acestor concepte vă va ajuta enorm în cariera de programator.

Mai departe vom învăţa să tratăm cu intrările şi ieşirile şi cum să accesam fişiere în Python.