

Cum să programezi în Python

(Manual în limba română)

Mircea Prodan

- 2014 -

Cuprins

Cuprins	2
De ce să te apuci de programare?	4
Cu ce limbaj ar trebui să încep?	5
De ce Python?	8
Capitolul I Instalarea Python 3. Noțiuni fundamentale	10
Instalarea Python3 în Windows	10
1.1. Primul program	12
1.2. Noțiuni fundamentale	14
1.3. Variabile	18
1.4. Puțină matematică	26
1.5. "Spargerea" declarațiilor lungi în linii multiple	28
1.6. Specificarea unui item separator	28
1.7. Concatenarea - afișarea mai multor itemi cu operatorul +	29
1.8. Formatarea numerelor (Important!!!)	29
1.9. Formatarea întregilor	33
1.10. Formatarea șirurilor	33
Capitolul II FUNCȚII	35
2.1. Definirea și invocarea unei funcții	35
2.2. Invocarea unei funcții	36
2.3. Indentarea (Important!!!)	37
2.4. Variabile locale	37
2.5. Constante și variabile globale	40
2.6. Funcția lambda	41
Capitolul III STRUCTURI DE DECIZIE	43
3.1. Declarația if	43
3.2. Operatori de comparație	44
3.3. Declarația if-else	44
3.4. Operatori logici	46
3.5. Variabile booleene	47
3.6. Bucle	47
3.7. Bucla while – o buclă controlată	47
3.8. Bucla infinită	49
3.9. Bucla for	49
3.10. Acumulatori	52
3.11. Operatori de atribuire augmentată	53
3.12. Bucle de validare a intrărilor	53
3.13. Bucle imbricate	55
Capitolul IV Module	56
4.1. Biblioteci de funcții standard și declarația import	56
4.2. Generarea numerelor aleatoare	56
4.3. Funcțiile randrange, random și uniform	58
4.4. Scrierea propriei funcții care returnează o valoare	59
4.5. Modularea cu funcții	60
4.6. Modulul matematic	61
Capitolul V Fișiere și excepții	64
5.1. Tipuri de fișiere	64
5.2. Metode de acces a fișierelor	65
5.3. Deschiderea unui fișier în Python	65

5.4. Scrierea datelor într-un fișier	66
5.5. Citirea datelor dintr-un fișier.....	68
5.6. Adăugarea datelor într-un fișier existent	70
5.7. Scrierea și citirea datelor numerice	70
5.8. Copierea unui fișier	71
5.9. Fișiere binare	72
5.10. Excepții.....	73
Capitolul VI Liste, tuple, dicționare și seturi. Serializarea obiectelor (pickling)	76
6.1. Liste.....	76
6.2. Metode și funcții preconstruite pentru liste.....	84
Metoda append	84
6.3. Tuple	89
6.4. Dicționare	90
6.5. Metode ale dicționarilor	95
6.6. Seturi.....	98
6.7. Serializarea obiectelor (pickling).....	100
Capitolul VII Clase și obiecte. Programarea orientată pe obiect.....	103
7.1. Definiții	103
7.2. Clase	106
7.3. Crearea claselor în Python	108
7.4. Adăugarea atributelor	109
7.5. Să <i>punem</i> clasa la treabă.....	110
7.6. Argumentul self.....	111
7.7. Definirea unei clase – o altă abordare	112
7.8. Moștenirea (Inheritance) în Python.....	117
Cap. VIII Crearea și manipularea formularelor web.....	123
8.1. Drepturi de acces la fișiere.....	131
Milioane de posibilități.....	133
Bibliografie.....	134

De ce să te apuci de programare?

Pentru că:

1. îți ține mintea trează
2. e fun
3. e o activitate care îți aduce prestigiu
4. poți câștiga bani (mulți!)

Până la 45 de ani nu mă interesa mai deloc IT-ul. Mi se părea un univers inaccesibil. Foloseam calculatorul ca orice *user* de rând: scriam texte pe care abia dacă reușeam să le "tehno-redactez", citeam sau trimiteam email-uri și uneori lecturam ziare sau publicații online. După aceea s-a produs declicul. Păcat că am descoperit acest minunat domeniu cam târziu, când creierul nu mai funcționează ca odinioară.

Toată viața am crezut că programarea înseamnă înainte de toate să fii tobă de matematică. E o exagerare. După ce am studiat limbaje de programare vreme de trei ani, am ajuns la concluzia că nu trebuie să știi mai mult de operațiile matematice fundamentale la care se adaugă și cunoașterea ordinii lor. Astea se învață din câte îmi amintesc prin clasa a șasea sau a șaptea, atunci când deslușești tainele algebrei ¹.

Cel mai important lucru în asimilarea unui limbaj de programare este în opinia mea **sintaxa**, adică regulile după care sunt puse semnele, numerele sau declarațiile dintr-un program. Din necunoașterea lor în profunzime vin și cele mai multe erori în scrierea unui program. Când am început să deslușesc tainele PHP uitam mai mereu să închei linia de program cu punct și virgulă așa cum e regula. Asta genera permanent erori pe care apoi, cu cât programul devine mai lăbărlat, le dibuiești greu, cam ca pe acul din carul cu fân.

Ce îți este necesar ca să scrii un program (pe lângă ceva cunoștințe de *mate* și stăpânirea sintaxei)? Dăruire, atenție și...scrierea de programe. Fiecare carte care afirma ca te învață un limbaj de programare are în general la sfârșitul capitolelor o secțiune de exerciții și probleme. E partea cea mai importantă (și captivantă)! Citiți bine teoria, scrieți

¹Asta desigur e valabil pentru un anumit nivel (mai lipsit de pretenții...) în programare.

programele date în carte în propriul editor de text (nu descărcați codul aferent cărții de pe site-ul editurilor pentru ca vă va fi lene să-l rescrieți), analizați-le și apoi procedați la rezolvarea exercițiilor. Numai așa înveți!

Cu ce limbaj ar trebui să încep?

Aici e ca în proverbul acela cu câte bordeie, atâtea obicei. Am stat odată ore întregi pe forumuri pentru a afla un răspuns cât de cât adecvat la această întrebare. Oricum ar fi, e bine să începi să studiezi un limbaj *ușor*. Dar există unul ușor, atât de ușor încât să fie asimilat într-o perioadă cât mai scurtă? Din experiența personală, nu. Totuși, un limbaj de programare foarte ușor de implementat pe calculator și nu prea greu de învățat este **Python**. Despre el am ales să scriu în această carte.

Există limbaje de programare complexe și altele mai puțin complexe. Există limbaje cu ajutorul cărora se pot construi doar aplicații web și altele care pot aduce la viață idei mărețe care ușurează viața de zi cu zi a celor care le folosesc. Acestea din urmă sunt cel mai greu de învățat dar odată asimilate îți garantează satisfacții majore. Limbaje de scriptare precum PHP sau Javascript sunt bune (doar) pentru a face site-uri, bloguri, aplicații pentru Internet în general. Când însă vorbim spre exemplu de programe de contabilitate, de proiectare în inginerie sau arhitectură, de aplicații IT în medicină sau științe atunci intervin *adevăratele* limbaje de programare: C, C++, Java, Python, Lisp, Pascal ² etc.

Sfatul meu este să începeți, dacă imi este permis, să studiați - dintre limbajele grele - pe cele mai ușoare! În niciun caz însă nu e indicat să vă apucați de învățarea C++. E adevărat, el este *regele balului* (alături de Java...), e extrem de utilizat dar foarte dificil de asimilat. Și C-ul este asemenea lui ținând cont că îi este precursor. Java pe de altă parte, seamănă mult (ca sintaxă cel puțin, cu toate că ceva mai...ușurică) cu C++ dar pe deasupra are un mediu de dezvoltare greu de configurat pentru un începător.

În tot cazul, nu vă apucați să deprindeți un limbaj de programare din sursă închisă precum cele furnizate de Windows sau Apple. E drept C# sau Objective C se folosesc (acum!) pe scară largă la construirea de aplicații pentru dispozitivele mobile aflate în mare

²Vezi site-ul Tiobe.com (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>)

vogă. Dar Țineți minte că *voga* e modă și de obicei ea trece – mai încet sau mai repede, dar tot trece. Sfatul meu e să învățați limbaje de programare portabile pe orice sistem de operare și mai ales din sursă deschisă. Unele dintre ele n-au moarte. C spre exemplu se îndreaptă spre 50 de ani³ și nu dă semne că va pieri în curând. C++ are și el peste trei decenii⁴ și la fel, e “în plină putere”, chiar dacă a mai scăzut în preferințele programatorilor.

Atunci când căutați materiale tipărite sau online, tutoriale, cărți, exemple etc pentru învățat, evitați titluri de genul : “Învăță Java în 24 de ore!” Sunt minciuni ordinare! Java, Python, C++ (chiar și Basic) **nu se învață** nici într-o luna, nici măcar într-un an, daramite într-o zi! Opinia mea – pe care o împărtășesc cu sinceritate, e că un limbaj de programare se asimilează în ani de zile. Și nici atunci, după o perioadă lungă de timp, nu cred că poți să pretinzi că ești “geek”...După unele voci de mare încredere⁵, perioada optimă de timp necesară deprinderii unui limbaj de programare este de zece ani!

Alegeți-vă un limbaj de programare pe care-l puteți iniția imediat și fără bătăi de cap la compilare. Cel mai indicat e desigur Python. Îl descarci într-o clipa de pe Internet, îi configurezi variabilele de mediu (în Windows) și dai drumul la lucru. Alta e treaba cu Java, C++ sau C. Ele necesită mai multă muncă și pentru un începător (care nu prea are pe cine să întrebe în afară de Google⁶...) nu e chiar ușor.

Marele noroc (astăzi!) în deslușirea unui limbaj de programare este că pe Internet se găsesc o mulțime de tutoriale și forumuri, locuri unde întotdeauna afli răspunsuri la întrebări pentru că alții s-au lovit înaintea voastră de aceleași probleme, desigur dacă ști să întrebi.

Uitasem un lucru esențial: **limba engleză**. Din observațiile pe care le tot fac de vreo câțiva ani, am observat că există câteva nații care excelează în IT. Printre ele, la loc de cinste, indienii, chinezii și nordicii. Am găsit și explicații. La indieni e vorba despre numărul uriaș al populației. Vă imaginați că din peste un miliard de oameni ai de unde

³ 1972

⁴ 1980

⁵ <http://norvig.com/romanian21-days.html>

⁶ Din păcate, literatura IT în România e sublimă dar (aproape) lipsește cu desăvârșire. Se mai găsesc rătăcite prin librării *vestigii* din anii '90 sau începutul ori mijlocul anilor 2000, total depășite astăzi.

alege câteva sute sau mii de căpățâni luminate. Aceștia au avut poate norocul educației și pe cale de consecință, a ieșirii din foame. Într-o țară în care speranța de viață e de 40 de ani, a fi instruit e o șansă uriașă! Programarea i-a scos pur și simplu din sărăcie, i-a adus adesea în America și i-a angajat la companii de vârf precum Windows, Oracle, IBM, Apple. Căutați informații despre diverse companii de top și nu se poate să nu găsiți în poziții de frunte indieni sau chinezi. Numărul foarte mare și sărăcia pe măsură este explicația succesului (multora dintre ei) în IT.

Ceilalți sunt nordicii. Câteva exemple edificatoare: cel care a proiectat Linux-ul este finlandez cu rădăcini suedeze; cel care a făcut inițial PHP-ul este danez, cel care a făcut C++ este danez, cel care a proiectat Python-ul este olandez. Și exemplele pot continua. Ei, contrar indienilor ori chinezilor, provin din țări foarte dezvoltate economic dar paradoxal, cu populație deficitară. Și atunci de unde formidabilul succes în IT? Am găsit și aici o explicație: condițiile meteo. Când ai jumătate sau mai bine din an temperaturi negative, când plouă sau ninge neîntrerupt zile întregi ești nevoit să stai în casă, eventual în fața computerului. Cu el îți omori timpul, cu el comunici cu alții și tot el te poate aduce pe culmile gloriei.

Și totuși, cei care au inventat calculatorul, limbajele inițiale de programare și tot ce ține de IT au fost **americani**, adică un popor care vorbește limba engleză, limbă în care logic, au scris și limbajele de programare, apropiate ca vocabular acesteia. De aici rezultă importanța fundamentală a limbii engleze în învățarea limbajelor de programare. Totuși, nu vreau să vă sperii. Nu trebuie să stăpâniți engleza la nivel *oxbridge*, însă un bagaj minim de cuvinte este absolut necesar să-l posedați. Mai trebuie să știți – fie doar pentru cultura generală – că la baza industriei IT de astăzi au stau odinioară proiecte militare americane (realizate în colaborare cu mediul academic), care datează încă hăt, din timpul celui De-al Doilea Război Mondial. Apoi a venit mediul de afaceri ce a avut nevoie de aplicații care să ușureze munca, să mărească productivitatea și implicit profitul. Ca o concluzie, IT-ul are origini militare, academice și financiare. Doar cei care puteau susține financiar cercetarea în domeniile de vârf puteau obține rezultate, nu? Timpul a dovedit că au reușit dar mai ales, aplicațiile – bazate 100% pe limbaje de programare și care odată, de mult, erau accesibile

doar unei mâini de oameni, au devenit astăzi un lucru banal și cât se poate de comun. Pentru asta trebuie doar să jonglați puțin cu un telefon mobil inteligent care este “înțesat” cu ele.

De ce Python?

Pentru că în primul rând este un limbaj curat ca sintaxă și foarte ușor de implementat pe orice calculator. Legenda spune că Guido van Rossum – creatorul Python, a pus bazele limbajului într-un week-end de Crăciun când se plictisea și nu prea avea cu ce să-și omoare timpul. Cert este că olandezul, care a lucrat mulți ani după aceea la Google⁷, a reușit să ofere lumii un limbaj cu o sintaxă simplă și suplă, cu reguli clare și de netrecut și care se folosește astăzi pe scară largă în mediul academic american (tocmai pentru învățarea limbajelor de programare), în afaceri dar și în proiectarea unor aplicații de mare succes. Youtube spre exemplu este scris în Python. Și Google folosește ca liant Python. Asemenea lui Yahoo!. Renumitul ziar Washington Post folosește în varianta lui online limbajul Python. Căutați pe Internet și veți găsi alte numeroase domenii în care Python are un cuvânt important de spus.

Un alt motiv pentru care este indicat să începeți cu Python este dat de numărul de entuziaști aflat într-o creștere constantă și care se ocupă de Python. Internetul este plin de informații despre el, de tutoriale, de cărți și forumuri. Este știut că într-un fel, Python nu poate să facă ceea ce face C++ spre exemplu. Totuși, curba de învățare a celor două limbaje este radical diferită. Cred sincer că un programator Python valoros se poate forma în circa doi – trei ani, bine-nțeles dacă stă cu brânca pe carte. Nu același lucru se poate afirma despre C++...

Altfel decât PHP – un alt limbaj relativ ușor de asimilat, cu Python se pot realiza proiecte web dar / sau mai ales, aplicații desktop de sine stătătoare.

Un alt motiv pentru care este bine să învățați Python e numărul relativ redus (pe plan mondial, în România ce să mai vorbim...) de programatori în acest limbaj. Știu, veți spune că site-urile de joburi sunt pline de oferte de muncă pentru programatori Java, C,

⁷ Van Rossum a plecat între timp la Dropbox, după șapte ani petrecuți în ograda Google.

C++, C# , PHP sau Objective C și mai puțin Python. Așa este, numai că numărul programatorilor Java sau PHP este covârșitor în vreme ce al celor care se ocupă de Python – cu toata creșterea lui - nu. De aici vine și confuzia care se creează cu privire la găsirea unui job legat de Python. Dați un banal *search* pe Google cu cuvintele cheie “python jobs(s)” și veți avea afișate instantaneu în fața ochilor un noian de rezultate care mai de care mai interesante. Sunt oferte peste oferte, ce-i drept, majoritatea de peste Ocean. Asta nu înseamnă că nu aveți șanse. Nu trebuie neapărat să lucrați *full time* la sediul din Chicago sau din Silicon Valey al unei firme, ci în intimitatea propriului cămin. Restul îl face Internetul, Skype, Messenger și/sau PayPal. Internetul a anulat distanțele, a făcut ca proiectele și ideile să circule nestingherite cu viteze uluitoare iar banii să intre (sau să iasă...) din conturi la fel.

Ultimul motiv și poate cel mai important este prestigiul. Stăpânirea Python (sau a oricărui alt limbaj de programare) face diferența în ochii cunoscătorilor, a prietenilor sau rudelor dar mai ales în ochii voștri. A ști să programezi îți ține mintea activă, te face să fii mândru de tine, să te simți tânăr și cine știe, cu multă muncă și ceva șansă, îți poate aduce și venituri substanțiale.

Succes!

Capitolul I Instalarea Python 3. Noțiuni fundamentale

În această carte vom lucra cu Python 3 (sau variantele lui ulterioare). În prezent există variantele Python 2 (și cele din clasa lui: 2.6 și 2.7 care sunt cele mai răspândite) și Python 3 (de asemenea cu variantele lui). Din păcate, Python 3 nu a devenit încă standardul dar, cel mai important, urmeaza sa devina! Tot din nefericire, majoritatea framework-urile necesare aplicațiilor web sunt scrise în Python2. Ele nu sunt compatibile cu Python 3, așa că dacă veți încerca să le instalați pe un computer care rulează Python 3 nu veți reuși. O excepție notabilă⁸ este din cele observate de mine, Pyramid. Oricum, până la a proiecta aplicații web scrise în Python e cale lungă de bătut pe care o începem cu..

Instalarea Python3 în Windows

Ca să programezi în Python trebuie mai întâi să-l instalezi pe computer. Îți alegi ultima distribuție de pe site-ul www.python.org. În [Windows](#) lucrurile sunt puțin mai complicate dar nu deznădăjduiți. Descarci de pe link-ul de mai sus (www.python.org) executabilul (.exe) ultimei variante pe care-l rulezi (Run). Pachetul este instalat automat pe partiția "C" (vă sfătuiesc să nu o schimbați). Dacă însă vei deschide o fereastră în linia de comandă și vei scrie *python*, limbajul nu va funcționa (încă) pentru că nu ai schimbat calea de rulare în *variabila de mediu* și calculatorul nu știe unde să-l găsească.

Iată care sunt pașii ca să faci asta:

1. Start -> Computer
2. Properties
3. Advanced Settings
4. Environment Variables
5. Path (Edit)
6. Aici adaugi la sfârșitul liniei următoarele: ; C:\Python32⁹ (nu uita de semnul punct și virgulă (;) scris *înaintea* datelor!)

⁸ Între timp și Django – cel mai folosit și cunoscut web framework Python, a migrat complet spre Python3 ori Mezzanine

⁹ python32 e varianta mea. La voi ar putea fi python33 sau python34 (varianta din momentul scrierii cărții)

7. Ok

8. Restartezi computerul

Dupa repornire, deschizi o nouă fereastră Command Prompt (DOS) și scrii *python*. Abia acum vei avea posibilitatea să “jonglezi” cu limbajul de programare Python în modul *interactiv*.

Ca să rulezi un program Python în modul *script* (vom vedea imediat ce înseamnă asta) în Windows 7 sunt necesari următorii pași:

1. Scrii programul în Notepad (sau orice alt editor, mai puțin MS Word..)
2. Îl salvezi (pe Desktop) cu extensia **.py** din meniul derulant al Notepad
3. Pornești Command Prompt-ul tastând cheile Windows+R și scriind în căsuță **cmd**, după care apeși Enter
4. Odată fereastra Command Prompt deschisă, schimbi aici directorul pe desktop cu comanda: `cd Desktop` (adică acolo unde ai salvat fișierul Python, să-i zicem `test.py`)
5. Scrii la prompter: `python test.py`

Dacă totul este în regulă și programul nu are erori de sintaxă, el va fi rulat fără probleme.

În MacOS X

Spre deosebire de Windows, în Mac OS X Python este deja (pre)instalat. Tot ceea ce trebuie să faci este să deschizi Terminal-ul (din Applications -> Utilities -> Terminal) și să scrii la prompt: *python*. Imediat îți va apărea prompterul python care este **>>>** și unde poți începe să... “programezi” (asta în modul interactiv). Dacă vei scrie însă programe într-un editor de text gen Smultron (până recent era gratis dar acum observ că dezvoltatorul i-a pus prețul de 5\$ pe Apple Store), atunci vei urma pașii de la rularea în Windows. În linii mari sunt aceiași. Scrii programul în editorul preferat, îl salvezi pe desktop cu extensia **.py**, deschizi Terminalul, schimbi directorul (`cd desktop`) și rulezi programul (`python test.py`).

De obicei OS X "Snow Leopard", "Lion", „Mountain Lion" și "Maverick" au deja preinstalate versiunile 2.6. sau 2.7. (ori mai vechi) ale Python. Dacă însă ai descărcat o variantă 3.0 sau peste (de exemplu 3.2. , 3.3. sau 3.4.), ea va fi instalată în *Applications*. Nu este nicio problemă, căci ca s-o rulezi trebuie doar ca în Terminal să scrii în linia de comanda `python3` și programul va începe să funcționeze. Totuși, dacă vrei ca ultima versiune Python (din clasa 3) să-ți pornească automat atunci când scrii doar `python` în Terminal (care îți va deschide invariabil varianta mai veche, cea preinstalată) trebuie să faci unele modificări.

Acestea sunt următoarele¹⁰:

1. În Terminal scrii: **`open ~/.bash_profile`** . Această comandă îți va deschide în editorul predefinit (TextEdit) fișierul pomenit.
2. Adaugi aici, în poziția cea mai de sus, următoarele:
3. **`alias python="python3"`**
4. Salvezi
5. Redeschizi Terminalul și scriind `python` vei observa că versiunea Python3 este cea care rulează.

1.1. Primul program

În Python există două moduri de a vizualiza programele:

- în interpretorul Python
- în modul script

Să vedem mai întâi, pentru a vă convinge că Python este un limbaj grozav, care este diferența între scrierea celui mai simplu program în Java și unul în Python. În Java:

```
// my first program in Java
```

¹⁰ În versiunea Mac OS Maverick, acest *truc* nu funcționează, astfel că în linia de comandă scrieți **`python3`** și dați enter. Dacă nu adaugați numărul **3** după comanda **`python`**, computerul afișează în Terminal shell-ul versiunii 2.7. care este preinstalată. Dacă lucrați în Ubuntu Linux, în versiunea 14 Python 3 este deja preinstalat. Scrieți în Terminal (shortcut „Ctr+Alt+T„) `python3` și este deschis shell-ul Python.

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }

}
```

și acum în Python:

```
>>> print('Hello World!')
```

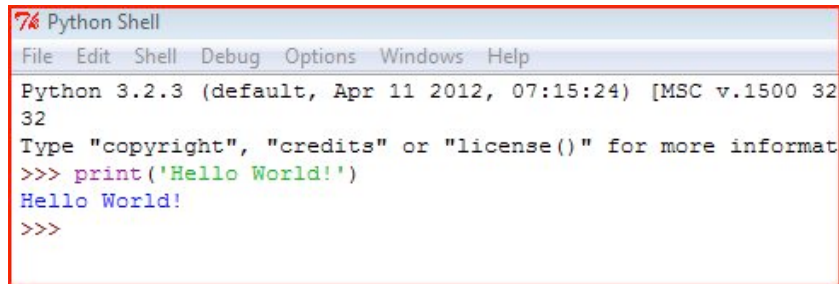


Fig.1.1. Hello World! în Python shell

Deci ceea ce în Java se afișează cu ajutorul a șase linii, în Python se face cu una singură! Pentru scrierea primului (și următoarelor) programe în Python trebuie să deschideți interpretorul Python astfel:

Windows -> All Programs -> Python32 -> Python (Command Line) [Enter]

Același lucru se face și fără invocarea interpretorului, direct din Command Prompt / fereastra DOS (Windows) sau Terminal (Mac OS, Linux). În fapt, sfatul meu este să lucrați doar cu acest instrument.¹¹

Există de asemenea și mediumuri profesionale de dezvoltare a programelor Python, cum ar fi Eclipse, Netbeans sau Pycharm (exclusiv pentru Python). Cu toate că primele două sunt dedicate programării în Java, ele admit instalarea de add-on-uri suplimentare pentru crearea și testarea programelor scrise în Python. Totuși, nu vă sfătuiesc să utilizați aceste mediumuri complexe (cel puțin nu la început de drum) care sunt îndeajuns de intimidante pentru un începător. În plus, nici configurarea lor inițială nu e prea lesne de realizat.

Să revenim însă la lucruri ușoare. Astfel, o dată ce ai scris o declarație aici (în Windows Command Prompt sau MacOS/Linux Terminal), automat – după apăsarea tastei “Enter”,

¹¹ Totul este însă ca în Environment Variables (Windows) calea de rulare către executabilul Python să fie configurată corect (vezi p.11-12 instalarea Python).

interpretorul Python o (și) afișează pe ecran:

```
Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 16 2013, 23:39:35)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Programarea în Python e frumoasa!")
Programarea în Python e frumoasa!
>>> █
```

Fig.1.2. Interpretorul Python în MacOS Terminal

1.2. Noțiuni fundamentale

Declarațiile (statements) sunt liniile (conținutul) unui program. Toate declarațiile dintr-un program scrise în modul script și salvate pe hard-ul calculatorului cu extensia **.py** se cheama *cod sursă* sau pur și simplu *cod*.

Modul script

Spre deosebire de interpretorul Python, modul script **salvează** declarațiile din program pe calculator. Acestea sunt de fapt, în opinia mea, *adevăratele* programe.

Să afișăm de exemplu cu ajutorul programului următor, câteva informații despre un personaj fictiv. Mai întâi scriem în editorul de text preferat (Notepad în Windows de exemplu sau Smultron în Mac OS) următoarele linii de cod:

```
print('Vasile Popescu')
print('Adresa: Str.N.Iorga, Nr.46, Bucuresti12')
print('Telefon: 0722 200406')
```

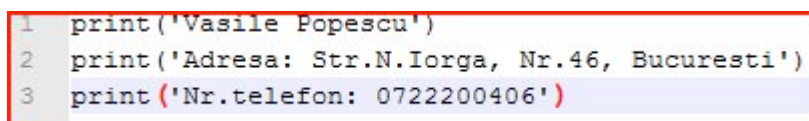


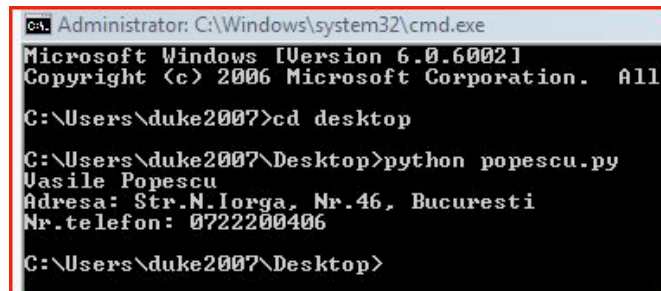
Fig.1.3. Programul scris în Notepad

Salvăm programul sub numele *popescu.py* (eu l-am salvat pe desktop) și apoi îl rulăm din linia de comandă cu comanda **python popescu.py**, nu înainte însă de a muta calea de rulare a programului pe desktop cu comanda **cd desktop¹³**. Altfel, interpretorul Python nu găsește fisierul (în cazul meu *popescu.py*) și va genera o eroare cât se poate de enervantă.

¹² Puteți la fel de bine să scrieți *București* (cu diacritice). Nu va rezulta nicio eroare pentru că Python folosește pentru decodificare standardul UTF-8 și nu ASCII.

¹³ Comanda `cd desktop` nu este *case sensitive* (nu ține cont dacă scrieți `desktop` sau `Desktop`) în Windows și MacOS dar e *case sensitive* în Linux. Astfel, dacă lucrați pe un computer cu sistemul de operare Ubuntu instalat (nu știu la celelalte distribuții Linux), comanda este neapărat `cd Desktop`, cu majusculă.

Poate vă întrebați de ce insist pe aceste aspecte care aparent nu par importante. Dimpotrivă, ele sunt foarte importante, mai ales la început, când știi doar să deschizi calculatorul dar esti plin de dorința de a învăța programare! Atunci când m-am apucat să deprind tainele programării, mă loveam mereu – și îmi venea să arunc computerul pe fereastră – de aceste amănunte care apar exact la început de drum!



```
ca. Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.0.6002]
Copyright (c) 2006 Microsoft Corporation. All
C:\Users\duke2007>cd desktop
C:\Users\duke2007\Desktop>python popescu.py
Uasile Popescu
Adresa: Str.N.Iorga, Nr.46, Bucuresti
Nr.telefon: 0722200406
C:\Users\duke2007\Desktop>
```

Fig.1.4. Rularea programului *popescu.py* în linia de comandă

Intrări, procesare, ieșiri, funcția print

Intrările (Input-ul) care se fac de obicei de la tastatură, sunt datele pe care computerul le primește de la utilizator. Urmează etapa procesării lor. Rezultatul operației de procesare se numește *ieșire* (Output) și este afișat pe ecran.

Afișarea ieșirii cu funcția print

O *funcție* este un cod prescris care realizează o operație. Python are numeroase funcții preconstruite. Totuși, dintre toate, fundamentală rămâne funcția **print**, adică exact aceea care afișează ieșirea pe ecran.

Când un program execută o funcție, spunem că el *cheamă* (invocă) funcția. Când invocăm funcția *print*, scriem cuvântul `print` urmat de un set de paranteze (). Înăuntrul parantezelor scriem *argumentul*¹⁴ care reprezintă datele pe care le dorim afișate pe ecran. Spre exemplu, la invocarea funcției `print` în declarația `print('Hello World')`, argumentul este **Hello World**. Observăm că ghilimelele nu sunt afișate la ieșirea programului. Ele doar arată începutul și sfârșitul textului pe care vrem să-l afișăm.

¹⁴ **Parametrul** este definit de numele care apare în definiția funcției, în vreme ce **argumentul** reprezintă variabila trecută unei funcții atunci când este invocată. Parametrul definește tipul de argument(e) pe care o funcție îl poate accepta.

Prin urmare, putem afirma că **print** este cea mai importantă funcție. Recapitulând, cu ajutorul ei afișăm ieșirea unui program în Python. Încercați în shell-ul Python următoarea linie de cod:

```
>>> print('Hello world!')
Hello world!
>>>
```

Șiruri și șiruri literale

Ex: `'Vasile Popescu'`
`'46, N.Iorga'`
`'Bucuresti, 0722200406'`

Acestea sunt șiruri de date care se mai cheamă *șiruri de date literale*. Ele sunt incluse între ghilimele. Se pot folosi ghilimele simple (' '), duble (" ") sau chiar triple (''' ''').

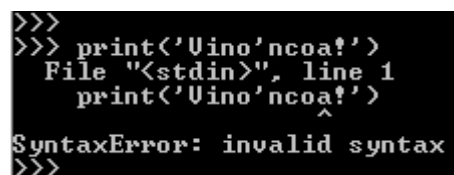
Să luăm de exemplu următoarea linie de cod care folosește ghilimele simple și duble:

```
print("Vino'ncoa!")
```

Ea va scoate la ieșire șirul literal:

Vino'ncoa!

Ce s-ar întâmpla dacă în expresia (șirul) de mai sus am pune *doar* ghilimele simple (la fel de bine duble sau triple)? Am primi din partea interpretorului (shell-ul) Python o eroare de sintaxă ca mai jos:



```
>>>
>>> print('Vino'ncoa!')
File "<stdin>", line 1
    print('Vino'ncoa!')
          ^
SyntaxError: invalid syntax
>>>
```

Fig.1.5. Eroare de sintaxă la punerea ghilimelor

Câteva cuvinte în plus trebuie adăugate despre **ghilimelele triple** (pe care nu le veți găsi în Java, C++, C ori PHP) dar care au o semnificație aparte în Python. Mulți folosesc ghilimelele triple pe post de comentarii, ceea ce este eronat. În Python comentariile sunt (*doar liniile de cod marcate*) cu semnul diez (#) la început (citiți paragraful următor).

De exemplu **#acesta este un comentariu.**

Ghilimelele triple sunt tratate ca *șiruri regulate*¹⁵ cu excepția cazului în care este vorba de mai multe linii. Totuși, ele nu sunt ignorate de interpretorul Python așa cum se întâmplă cu comentariile care încep cu semnul diez (#). Ele sunt așezate imediat după definiția unei funcții sau clase, ori în vârful codului unui modul, caz în care se numesc **docstrings**. Docstring-urile pot include în interiorul lor ghilimele simple, duble sau de ambele feluri ca-n exemplul de mai jos:

```
>>>
>>> ghilimele_triple='''Io mi's "Aeropagul"
... zise Gigel speriat'''
>>> print(ghilimele_triple)
Io mi's "Aeropagul"
zise Gigel speriat
>>> █
```

Fig.1.6. Utilizare ghilimele triple

Să vedem acum ce se întâmplă dacă scriem doar *ghilimele_triple* (fără funcția `print`) și dăm enter:

```
>>>
>>> ghilimele_triple='''Io mi's "Aeropagul"
... zise Gigel speriat'''
>>> ghilimele_triple
'Io mi\'s "Aeropagul"\nzise Gigel speriat'
>>> █
```

Fig.1.7. Un șir „ciudat” de caractere apare la finalul procesării programului

Răspunsul este ușor nefiresc pentru că el ne arată codificarea intimă în Python folosind caracterul „escape” (\).

Comentariile

Sunt folosite într-un program ca să arăți ce ai vrut să faci acolo. Dacă peste câțva timp vei redeschide programul, comentariile îți vor reaminti de ceea ce ai vrut să faci cu linia de cod respectivă din program. Ele sunt ignorate de interpretorul Python dar nu trebuie ignore de tine!

În Python, comentariile încep cu semnul diez #.

¹⁵Șirurile regulate sunt șirurile care **nu sunt** atribuite unei variabile.

```
Ex:  #Acest program afiseaza
      #numele unei persoane
      print('Mircea Prodan')
```

1.3. Variabile

Programarea înseamnă înainte de toate a jongla cu abstracțiuni. A lucra cu astfel de entități ce nu sunt palpabile – cel puțin nu în faza lor de dezvoltare, este în opinia mea cel mai dificil lucru pentru că în general, marea noastră majoritate avem o putere destul de limitată de a imagina obiecte pipăibile dar care au drept fundament idei „incolore”, „inodore” și mai cu seamă fără o formă anume. Din acest motiv ni se pare grea matematica...

Variabilele sunt printre cele mai importante elemente din programare și nu mă feresc să afirm că dacă ele și manipularea lor este bine înțeleasă, ați făcut un mare pas înainte în dobândirea unui limbaj de programare.

Totuși, spre deosebire de lumea programării orientată pe obiect (OOP), înțelegerea variabilelor este mulțumitor de ușoară. Tot ceea ce trebuie să faceți este **să vă gândiți la o variabilă ca la o cutie în care depozitați ceva.**

Permiteți-mi s-o luăm altfel. Să presupunem că ați terminat recent facultatea, ați redactat licența – care v-a fost aprobată de profesorul îndrumător și apoi, așa cum cere regulamentul, trebuie s-o puneți (s-o depozitați) pe un CD ce urmează să stea la arhiva universității multe decenii de atunci încolo, dar și să predați un exemplar tipărit. Lucrarea dumneavoastră de licență – nu importă domeniul – este pe calculatorul personal sub forma unui fișier Microsoft Word¹⁶ și deci ea este abstractă. Puteți s-o țineți în mână? Nu, eventual puteți s-o arătați cu degetul pe monitorul computerului! În momentul în care o printați și legați, ea își pierde această calitate și devine un obiect în sine, cât se poate de palpabil, cu care eventual vă puteți lăuda la familie și prieteni. Mergeți apoi la facultate și depuneți un exemplar printat. Ea este înregistrată și depozitată într-o încăpere a facultății. Hopa! Iată că dintr-o dată avem **variabila** care este *încăperea* și **valoarea** ei care este *lucrarea de licență*. Pe

¹⁶ Ar putea la fel de bine să fie o aplicație de redactare din sursă deschisă în genul OpenOffice ori LibreOffice, numai că universitățile românești acceptă doar fișiere Microsoft Word.

măsură ce alți și alți (potențiali) absolvenți își depun lucrările, numărul (valoarea) lor crește dar încăperea (variabila¹⁷) rămâne aceeași.

Mutându-ne oarecum în lumea computerelor, aveți acum nevoie și de CD-ul care să conțină lucrarea în format digital. Prin urmare “ardeți” un CD nou nouț, depozitând pe/în el lucrarea în format digital. Aici **variabila** este CD-ul căruia i-am atribuit **valoarea** *lucrare de licență*. Exemplul meu poate nu este cel mai fericit pentru că un CD obișnuit se poate inscripționa doar o singură dată, ceea ce ar însemna că o dată atribuită o valoare variabilei, aceasta nu se mai poate modifica. **În programare acest fapt nu este adevărat** (din acest motiv se numește *variabilă*). Totuși, există CD-uri care se pot inscripționa de mai multe ori, radiind informațiile (valorile) de la un moment dat și înlocuindu-le cu altele. Numele variabilei (în cazul nostru CD) rămâne același, doar valoarea pe care i-o atribuim (*licența*) se schimbă.

O variabilă este un nume (cuvânt) care reprezintă o valoare stocată în memoria calculatorului. Numele (variabila) se scrie întotdeauna în partea stângă:

```
age = 49  #Corect! Vârsta mea la data scrierii #primei
variante a lucrării

și nu

49 = age  #Gresit!!

age = 52  #Corect!Vârsta mea la data scrierii #variantei
actuale

și nu

52 = age  #Gresit!!
```

Semnul = nu înseamnă egal ca în matematică ci reprezintă **atribuirea**.

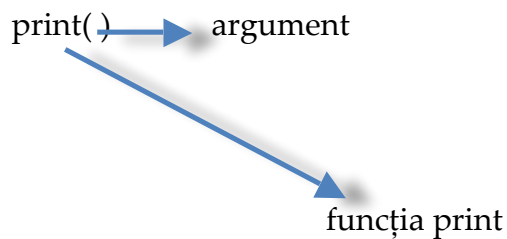
În exemplul de mai sus :

(lui) age = (i-am **atribuit** valoarea) 49 respectiv 52.

Când treci o variabilă ca argument al funcției print nu mai ai nevoie de ghilimele.

Ex:

¹⁷ De fapt numele ei...



```
>>>latime = 10  (Enter)
>>>print('latime')
latime
>>> print(latime) (Enter)
10
>>>
```

Exemplu de program care demonstrează o variabilă:

(*variabila.py*)

```
#Acest program demonstreaza o variabila
camera = 123
print('Stau in camera: ')
print(camera)
```

Ieșirea (output-ul) acestui program este:

```
Stau in camera:
123
```

Sa vedem in linia de comanda (Terminal) cum putem face economie de spațiu și efort prin renunțarea la ultima declarație `print`:

```
>>> camera=123
>>> print('Stau in camera ', camera)
Stau in camera  123
>>> █
```

Fig.1.8. Variabila șir

Ce s-ar fi întâmplat dacă în prima varianta a exemplului anterior, în ultima declarație `print` am fi cuprins între ghilimele parametrul `camera` ca mai jos? Păi în loc să ne afișeze valoarea parametrului care este **123**, ne-ar fi printat cuvântul (șirul literal) **camera**:

```

>>> camera=123
>>> print('Stau in camera ', camera)
Stau in camera 123
>>> print('camera')
camera
>>> █

```

Fig. 1.9. Importanța ghilimelelor¹⁸ la afișarea rezultatelor

Ați sesizat diferența?

Exemplu de program cu două variabile:

(*variabila2.py*)

```

#Creeaza doua variabile numite viteza si distanta
viteza = 160
distanta = 300
#Afiseaza valoarea referita de variabile
print ('Viteza este: ')
print (viteza)
print ('Distanta parcursa este: ')
print (distanta)19

```

Este indicat ca numele variabilei să înceapă cu literă mică. Variabilele sunt sensibile la litere mici sau mari (*case sensitive*) așa că de exemplu *python* nu este totuna cu *Python*.. Numele variabilelor nu au voie să conțină cuvintele cheie din Python²⁰. Variabilele nu trebuie să conțină spații dar au voie să conțină (și să înceapă) cu semnul underscore (_). De asemenea, variabilele nu trebuie să înceapă cu un număr sau să conțină caractere speciale precum \$ # % ^ & * ș.a.m.d.

Cu funcția `print` se pot afișa mai mulți itemi. Ex. (refacerea exemplului *variabila.py*):

```

#Acest program demonstreaza o variabila

```

¹⁸ Se pot folosi ghilimele simple sau duble, rezultatul este același.

¹⁹ Încercați acest exemplu pe calculatorul dumneavoastră, dar să cuprindă două declarații `print` și nu patru.

²⁰ ['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

```
camera = 123
print('Stau in camera numarul ', camera)
```

Ieșirea este: *Stau in camera 123*

Dacă priviți cu atenție, veți observa ca propoziției îi lipsește ceva esențial (ca să îndeplinească normele gramaticale de definire a propoziției) și anume punctul de la sfârșit. Haideți să vedem cum îl afișăm. (Încercați să figurați răspunsul fără să citiți paragraful de mai jos).

Dacă îl punem imediat după ultimul parametru al funcției `print (camera)` va rezulta o eroare de sintaxă. Pentru a nu primi o eroare, punctul trebuie la rândul lui cuprins între ghilimele ca în exemplul de mai jos:

```
>>> print('Stau in camera ', camera)
Stau in camera 123
>>> print('Stau in camera ', camera, '.')
Stau in camera 123 .
>>> print('Stau in camera ', camera, '.', sep='')
Stau in camera 123.
>>> █
```

Fig1.10. Afișarea corectă a unei declarații

Să mai observăm (în linia 4) că dacă nu adăugăm un *item separator*²¹ (ca-n finalul liniei 5), punctul este așezat ușor anormal, la o distanță destul de mare de locul în care ar trebui să fie. În output-ul din linia 6 această mică inadvertență (corectă totuși din punct de vedere al programării, nu însă și estetic) este eliminată.

Stocarea șirurilor cu tipul de date str

#Creeaza o variabila care refera doua siruri

```
prenume = 'Vasile'
```

```
nume = 'Popescu'
```

#Afiseaza valorile referite de variabile

```
print(prenume, nume)
```

La ieșire vom avea: *Vasile Popescu*

²¹ Am luat-o puțin înainte, dar veți vedea foarte curând ce reprezintă un item separator.

La fel de bine putem să inversăm numele dacă ultima declarație ar arăta astfel:

```
print(nume, prenume)
```

iar la ieșire ar rezulta *Popescu Vasile*.

Tipuri de date. Reatribuirea variabilelor diferitelor tipuri

În Python există câteva tipuri principale de date:

1. `int` - întregi
2. `float` - în virgulă mobilă²²
3. `string (str)` - șiruri
4. `boolean` – `True`, `False`

Ca să aflăm cu ce fel de date lucrăm utilizăm funcția **`type ()`** ca în exemplele de mai jos:

```
>>> type(45)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('Marcel')
<class 'str'>
>>> x = True
>>> type(x)
<class 'bool'>
>>> █
```

Fig.1.11. Folosirea funcției `type()` pentru aflarea tipurilor de date

Ce se întâmplă însă dacă unui întreg îi adăugăm zero la sfârșit? Pentru noi el rămâne un întreg, nu însă și pentru calculator. Astfel, el devine un număr în virgulă mobilă (`float`):

```
>>> type(45)
<class 'int'>
>>> type(45.0)
<class 'float'>
>>> █
```

Fig.1.12. Transformarea `int` în `float`

În Python, o variabilă poate să refere orice tip de date: întreg, șir, în virgulă mobilă (`float`), boolean. După ce o variabilă a fost atribuită unui anumit tip, ea poate fi reatribuită altui tip de date.

²² Acestea nu sunt totuna cu numerele zecimale

Ex:

```
>>> x = 99 [enter] #aici avem o variabila int (întreg)
```

```
>>> print(x) [enter]
```

99

dar și :

```
>>> x = 'Sa mergem la masa' (enter) #aici avem o variabilă #str (șir)
```

```
>>> print(x) [enter]
```

Sa mergem la masa

```
>>>
```

sau float:

```
x = 0.15
```

```
print(x) [enter]
```

0.15

Putem de fapt să renunțăm la 0 și să lăsăm doar punctul urmat de valorile semnificative:

```
>>> x = .15
>>> print(x)
0.15
>>> █
```

Fig.1.13. Am renunțat la 0 în declararea variabilei subunitare

Se observă că rezultatul este afișat însă normal – cu 0 înainte.

Obținerea inputului de la user

În programare, obținerea informațiilor de la user este extrem de importantă. Altfel ca în Java de exemplu, în Python acest lucru se face foarte ușor cu ajutorul funcției `input()`. Prompterul care clipește în fereastra DOS sau în Terminal este de fapt un șir care invită userul să introducă date de la tastatură.

Ex.:

```
name = input('Care este numele tau? ')
```

variabilă tastatură datele șir aici este un spațiu!!

Spațiul dintre semnul întrebării din șirul 'Care este numele tău?' și ultima ghilimeă este necesar pentru ca după răspuns, întrebarea și răspunsul să nu fie lipite (de ex: *Care este numele tău?Vasile Popescu* în loc de *Care este numele tau? Vasile Popescu*).

Exemplu de program cu două șiruri la introducerea de la tastatură:

```
#Ia prenumele userului
prenume = input('Introdu prenumele: ')
#Ia numele de familie
nume = input('Introdu numele de familie: ')
#Afiseaza numele userului
print ('Salut', prenume, nume)23
```

Citirea numerelor cu funcția input

Funcția **input** returnează întotdeauna *input*-ul (intrarea) userului ca pe un șir, chiar dacă userul introduce date numerice.

De exemplu, dacă în fereastra interactivă tastezi numărul 65 și apeși Enter, valoarea returnată de funcția **input** este șirul '65'. Aceasta poate fi o problemă dacă vrei să folosești operații matematice.

Așadar:

Operațiile matematice pot fi făcute doar cu valori numerice și NU cu șiruri.

Exemplul următor folosește funcția **input** ca să citească un șir (str), un numar întreg (int) și unul în virgulă mobilă (float).

(*input.py*)

```
#Ia numele, varsta si venitul financiar al userului
nume = input('Care e numele tau? ')
varsta = int(input('Cati ani ai? '))
venit = float(input('Care e venitul tau? '))
#Afiseaza datele
print('Iata datele pe care le-ai introdus: ')
print('Nume', nume)
```

²³ **Reproduceți** toate aceste exemple încet și atent pe computerul dumneavoastră. Nu le copiați ca să vedeți cum și dacă funcționează!! Programarea înseamnă două lucruri esențiale: pasiune și răbdare...

```
print('Varsta', varsta)
print('Venit', venit)
```

Concluzia este următoarea: **atunci când introduci nume sau alte date literale scrii simplu input ('.....').** Când însă introduci numere (întregi sau în virgulă mobilă) e musai să **arăți de ce tip sunt (int sau float)**²⁴.

`int` și `float` funcționează doar dacă itemul ce urmează să fie introdus conține o valoare numerică. Altfel, va apărea o eroare numită *excepție*.

1.4. Puțină matematică

Python are numeroși operatori cu care se pot face calcule matematice. Operatorii sunt aceiași ca în matematica: `+` `/` `*` `-` dar și:

`//` împărțirea cu întreg (rezultatul e totdeauna un întreg, de ex. $10 // 3 = 3$)

`%` rest – împarte un număr la altul și atribuie variabilei restul

`**` ridicarea la putere a unui număr

(și celalalte simboluri matematice).

Ex.

```
>>> 10//3
3
>>> 10%3
1
>>> 10**3
1000
>>> █
```

Fig.1.14. Împărțirea cu întreg, restul și ridicarea la putere

Să notăm că la ridicarea la putere, primul număr (în exemplul de mai sus 10) este cel multiplicat la puterea dată de cel de-al doilea număr (3 în cazul nostru).

Ex. *salariu.py*

```
#Atribuie o valoare variabilei salariu
salariu = 2500.0
```

²⁴ Cei care au programat în Java sau C știu că acolo e obligatoriu să declari tipul variabilei. De fapt, același lucru se petrece și în Python, dar doar în cazul datelor cu valoare numerică.

```
#Atribuire o valoarea bonusului
bonus = 1200.0
#Calculeaza venitul total si
#atribuie valoarea variabilei plata
plata = salariu + bonus
#Afiseaza plata
print('Plata ta este: ', plata)
```

Calcularea unui procent

Algoritmul este următorul:

- ia prețul original al unui item
- calculează 20% din prețul original; acesta este discountul
- scade discountul din prețul original; acesta e prețul de vânzare
- afișează prețul

Și acum să vedem programul (*pret_vanzare.py*):

```
#Acest program ia pretul original si
#calculeaza un discount de 20%.
#Ia pretul original
pret_original = float(input('Introdu pretul original: '))
#Calculeaza valoarea discountului
discount = pret_original * 0.2
#Calculeaza pretul de vanzare
pret_vanzare = pret_original - discount
#Afiseaza pretul de vanzare
print('Pretul de vanzare este: ', pret_vanzare)
```

Precedența operatorilor matematici

1. Ridicarea la putere se execută prima (**)
2. Înmulțirea, împartirea și restul (* / // %) al doilea
3. Adunarea și scăderea ultimele

Ridicarea la putere a unei valori se scrie așa:

Ex: `suprafata_patrat = lungimea**2`

în care `**2` este ridicarea la puterea a doua

sau

`volum_cub = lungimea**3` (ridicarea la puterea a treia)

Operatorul rest (remainder) %

Ex. `rest = 17 % 3`

Variabilei rest îi atribuim valoarea 2 pentru ca $17/3 = 5$ rest 2.

1.5. “Spargerea” declarațiilor lungi în linii multiple

Python permite *să spargi* o declarație în linii multiple folosind backslash-ul (`\`).

Ex:

```
print('Am vandut', bucati_vandute, 'pentru un total de', vanzare_totala)
```

devine

```
print ('Am vandut', bucati_vandute, \
      'pentru un total de', vanzare_totala)
```

Atunci când se sparge o declarație considerată prea lungă, **indentarea** (vom vedea puțin mai târziu ce înseamnă acest lucru) este obligatorie.

1.6. Specificarea unui item separator

Când argumentele sunt trecute funcției `print`, ele sunt automat separate de un spațiu care este afișat pe ecran.

Ex

```
>>> print('Unu', 'Doi', 'Trei') [enter]
```

```
Unu Doi Trei
```

```
>>>
```

Dacă le vrei împreunate, treci argumentul `sep=' '` funcției `print` la sfarsitul declaratiei:

```
>>> print('Unu', 'Doi', 'Trei', sep=' ')    #[enter]
```

```
UnuDoiTrei
```

>>>

```
>>>
>>> print('Unu','Doi','Trei', sep='')
UnuDoiTrei
>>>
```

Fig.1.15. Inserarea itemului separator

Caracterul escape

Caracterul *escape* este unul special care e ilustrat de un backslash (\) ce apare înăuntrul unui șir literal.

Ex.

```
print('Unu\nDoi\nTrei')
```

afișează:

Unu

Doi

Trei

ceea ce înseamnă că **\n** reprezintă o linie nouă (n vine de la *new line*)²⁵.

1.7. Concatenarea - afișarea mai multor itemi cu operatorul +

Când operatorul + se folosește la adunarea șirurilor se cheamă **concatenare** (legare).

Ex.

```
print('Acesta este'26 + 'un sir.')
```

care afișează:

Acesta este un sir.

1.8. Formatarea numerelor (Important!!!)

Când un număr în virgulă mobilă e afișat pe ecran, el poate avea și 12 zecimale.

Ex:

```
#Acest program demonstreaza cum un numar
```

²⁵ Încercați pe calculatorul personal să înlocuiți \n cu \t și vedeți ce rezultă. (Indiciu: litera t vine de la tab)

²⁶ Nu uitați că între ultimul cuvânt din primul enunț și ultima ghilime care încadrează același enunț **este un spațiu**. Dacă nu e marcat corespunzător, ultimul cuvânt din primul enunț și primul din cel de-al doilea vor fi afișate lipite.

```
#in virgula mobila este afisat fara formatare
```

```
suma_datorata = 5000.0
```

```
plata_lunara = suma_datorata / 12.0
```

```
print ('Plata lunara este ', plata_lunara)
```

iar ieșirea programului este:

```
Plata lunara este 416.666666667
```

```
>>> suma_datorata=5000.0
>>> plata_lunara=suma_datorata/12.0
>>> print('Plata lunara este: ', plata_lunara)
Plata lunara este: 416.6666666666667
>>> █
```

Fig.1.16. Valoare neformatată

Cum facem totuși ca rezultatul să nu mai apară ca o înșiruire grosiera de numere? Simplu: invocăm funcția **format**. Când invoci funcția **format**, îi treci două argumente:

- o valoare numerică
- un specificator de format

Specificatorul de format (*format specifier*) este un șir care conține caractere speciale ce arată cum valorile numerice trebuie formatate.

Ex: **format (12345.6789, '.2f')**

Primul argument, care este un număr în virgulă mobilă (12345.6789), este numărul pe care vrem să-l formatăm.

Al doilea argument este un șir – **' .2f '** – și reprezintă *specificatorul de format*.

Iată ce înseamnă el luat “pe bucăți” :

.2 specifică precizia; el arată că vrem să rotunjim numărul la două zecimale.

f vine de la float și specifică faptul că numărul pe care-l formatăm este în virgulă mobilă (pentru formatarea întregilor, cum vom vedea, nu se folosește litera f ci d).

În aceste condiții, funcția **format** returnează un șir care conține numărul formatat:

```
>>> print(format(12345.6789, '.2f')) #[Enter]
```

```
12345.68
```

Să notăm că numărul este rotunjit la două zecimale, în sus²⁷.

Acuma, să luăm același exemplu, dar rotunjit la o singură zecimală:

```
>>> print(format(12345.6789, '.1f')) [Enter]
12345.7
```

Sa luăm un alt exemplu în fereastra interactivă:

```
>>> print('Acesta este: ', format(416.66666666666666, '.2f'))
Acesta este: 416.67
>>> █
```

Fig.1.17. Valoare formatată cu specificatorul de format

Formatarea în mod științific

Se utilizează literele e sau E în loc de f.

Ex:

```
>>> print(format(12345.6789, 'e'))
1.2345678e +04
>>> print(format(12345.6789, '.2e'))
1.23e + 04
>>>
```

Inserarea separatorului virgulă

```
>>> print(format(123456789.456, ',.2f'))
123,456,789.46
```

separatorul virgulă²⁸

Programul următor demonstrează cum separatorul virgulă și o precizie de două zecimale pot fi folosite la formatarea unui număr mare:

(plata_anuala.py)

²⁷ Pentru că a doua valoare din șirul de numere de după punct - în speță 7 (.6789) este mai aproape de 10 decât de 0, computerul rotunjește valoarea *în sus*, ea devenind .68. Dacă ar fi fost în schimb .64..., .63..., .62..., .61... atunci era rotunjită *în jos*, ea devenind 0.60.

²⁸ Între separatorul virgulă și punctul situat înaintea lui (.2f) **nu există** spațiu. Dacă este introdus din greșală va rezulta o eroare ("Invalid format specifier").

```
#Acest program demonstreaza formatarea
#numerelor mari
plata_lunara = 5000.0
plata_anuala = plata_lunara * 12
print('Plata ta anuala este $',\
      format(plata_anuala, ',.2f'), \
      sep='')
```

Ieșirea programului este:

Plata ta anuala este \$60,000.00

Sa vedem cum funcționează programul în Terminal:

```
>>> plata_lunara=5000.0
>>> plata_anuala=plata_lunara*12
>>> print('Plata ta anuala este $',\
...       format(plata_anuala, ',.2f'), \
...       sep='')
Plata ta anuala este $60,000.00
>>> █
```

Fig.1.18. Rularea programul *plata_anuala.py* în shell-ul Python

Să notăm că în ultima linie a codului am trecut `sep=' '` ca argument al funcției `print`. Acesta specifică faptul că nu trebuie afișat niciun spațiu între itemii care urmează să fie afișați. Dacă nu facem acest lucru, va apărea un spațiu între \$ și sumă.

Specificarea unei lățimi minime de spațiu

Următorul exemplu afișează un număr în câmpul care este de lățime 12:

```
>>> print('Numarul este', format(12345.6789, '12,.2f'))
Numarul este    12,345.68
>>>
>>> print('Numarul este', format(12345.6789, '12,.2f'))
Numarul este    12,345.68
>>> █
```

Fig.1.19. Specificare lățimii de spațiu

Formatarea unui număr în virgulă mobilă ca procent

Aici în loc să folosim litera **f** folosim simbolul procentului (**%**) ca să formatăm un număr ca

procent:

```
>>> print(format(0.5, '%'))  
50.000%
```

Și încă un exemplu care are 0 ca precizie:

```
>>> print(format(0.5, '.0%'))  
50%
```

1.9. Formatarea întregilor

Diferențele la formatarea întregilor față de numerele reale (în virgulă mobilă) sunt:

- folosești **d** în loc de **f**
- NU poți specifica precizia

Ex:

```
>>> print(format(123456, 'd'))  
123456
```

Acuma, același exemplu dar cu separatorul virgulă:

```
>>> print(format(123456, ',d'))  
123,456
```

1.10. Formatarea șirurilor

Formatarea șirurilor e puțin mai complicată și se face cu ajutorul acoladelor ca în exemplul următor:

```
>>>  
>>> nume = input('Introdu numele tau: ')  
Introdu numele tau: Mircea  
>>> salut = 'Salut {}!'.format(nume)  
>>> print(salut)  
Salut Mircea!  
>>> █
```

Fig.1.20. Exemplu de formatare a șirurilor

Șirurile și alte obiecte au o sintaxă specială pentru funcții numită *metodă*, asociată unui tip particular de *obiect*. Obiectele de tipul șir (str) au o metodă numită *format*. Sintaxa pentru această metodă conține obiectul urmat de punct urmat de numele metodei și următorii parametri dintre paranteze:

`obiect.nume_metoda(parametri)`

În exemplul de mai sus obiectul este șirul `'Salut {}'` iar metoda este `format`.

Parametrul este `nume`.

După cum se observă la ieșire, acoladele sunt înlocuite de valoarea preluată din lista parametrilor metodei `format`. Deoarece acoladele au un înțeles special în formatarea șirurilor, este nevoie de o regulă specială dacă vrem ca ele să fie incluse în formatarea finală a șirurilor. Regula este **dublarea** acoladelor: `'{{' și '}}'`

De exemplu²⁹:

```
>>> a=5
>>> b=9
>>> formatSir='Setul este {{{}, {}}}.'
>>> setSir = formatSir.format(a,b)
>>> print(setSir)
Setul este {5, 9}.
>>>
```

Fig. 1.21. Formatarea șirurilor

²⁹ Vom vedea mai târziu ce reprezintă un *set* în Python.

Capitolul II FUNCȚII

O funcție reprezintă un grup de declarații care există într-un program pentru a realiza o anumită sarcină. Am văzut în primul capitol comportamentul funcției prestabilite `print`.

Majoritatea programelor realizează sarcini care sunt îndeajuns de mari ca să poată fi divizate în câteva subsarcini. Din acest motiv programatorii “sparg” programele în bucăți mai mici cunoscute sub numele de **funcții**.

Astfel, în loc să scriem largi secvențe de declarații, scriem câteva funcții mai mici, fiecare realizând o parte specifică din sarcină.

Aceste mici funcții pot fi executate în ordinea dorită pentru ca în final să realizeze soluția la întreaga problemă.

2.1. Definirea și invocarea unei funcții

Definirea unei funcții se face cu ajutorul cuvântului **def** (define). Codul unei funcții este (și) definiția unei funcții. Ca să execuți o funcție, scrii pur și simplu declarația care o invocă (numele funcției) urmată de paranteze `()`.

Numele funcțiilor

Numele unei funcții e bine să fie apropiat de ceea ce ea face. În Python, când denumim o funcție urmăm aceleași reguli ca la botezarea *variabilelor* adică:

- nu poți folosi cuvintele cheie din Python
- nu pot exista spații între cuvintele care alcătuiesc numele funcției
- primul caracter trebuie să fie o literă de la *a* la *z*, de la *A* la *Z* sau underscore (`_`)
- literele mici sunt tratate diferit de cele mari (case sensitive).

Pentru că funcțiile realizează acțiuni, e indicat să folosim verbe atunci când le alegem numele.

De exemplu, o funcție care calculează venitul poate fi denumită pur și simplu `calculeaza_venitul`. Unii programatori folosesc pentru notarea funcțiilor metoda numită “cocoașă de camilă” (camelCase). Dacă am folosi-o în exemplul nostru, atunci ea ar

arăta așa: calculeazaVenitul.

Formatul general este:

```
def nume_functie() :  
    declaratie  
    declaratie  
    .....
```

Ex:

```
def mesaj() :  
    print('Sunt Gigel,')  
    print('si incerc sa invat Python!')
```

Codul de mai sus definește o funcție numită *mesaj*. Funcția *mesaj* conține un bloc cu două declarații.

2.2. Invocarea unei funcții

Definiția unei funcții specifică ce face funcția dar nu o execută. Ca să execuți o funcție trebuie să o invoci (să o chemi).

Iată cum invocăm funcția *mesaj* din exemplul anterior:

```
mesaj()
```

Când funcția e chemată, interpretorul sare la acea funcție și execută declarațiile din blocul ei. Apoi, când blocul s-a terminat, interpretorul sare înapoi la partea de program care a chemat funcția și programul rezumă execuția în acel punct. Când se întâmplă asta, spunem că funcția *returnează*.

Ex:

(*functie_demo.py*)

```
#Acest program defineste o functie                                1  
#Prima data definim o functie numita mesaj                        2  
def mesaj() :                                                    3  
    print('Sunt Mircea,')                                         4  
    print('si incerc sa invat Python.')                          5  
#Apoi invocam functia mesaj                                       6  
mesaj()                                                           7
```

Cum funcționează? Interpretorul ignoră comentariile. Apoi citește declarația def în linia 3.

Aceasta creează o funcție numită `mesaj` în memorie. Ea conține declarațiile din liniile 4 și 5. Apoi interpretorul citește comentariul din linia 6 care este ignorat. La sfârșit citește linia 7 care invocă funcția `mesaj`. Aceasta face ca funcția să fie executată și să fie afișată ieșirea.

```
>>> def mesaj():
...     print('Sunt Mircea')
...     print('si incerc sa invat Python.')
...
>>> mesaj()
Sunt Mircea
si incerc sa invat Python.
>>> █
```

Fig. 2.1. Ilustrarea unei funcții în Python shell

2.3. Indentarea (Important!!!)

Într-un bloc indentarea se face cu același număr de linii. De obicei se folosește `tab-ul` sau patru spații. Dacă în același program folosim o dată patru spații , altădată `tabul` sau trei, cinci ori "n" spații, programul va da eroare de indentare:

```
>>> def mesaj():
... print("Va rezulta o eroare de indentare")
  File "<stdin>", line 2
    print("Va rezulta o eroare de indentare")
    ^
IndentationError: expected an indented block
>>> █
```

Fig.2.2. Eroare de indentare

Indentarea în Python este obligatorie și se face pentru ca blocurile de cod din program să fie perfect stabilite, ceea ce duce la lizibilitatea mai bună a programului. Indentarea ține de fapt locul acoladelor din celelalte limbaje de programare din familia limbajului C. Ea ne arata unde începe și se termină un bloc de declarații. Totodată ține loc și de semnul punct și virgula cu care se sfârșesc obligatoriu declarațiile din limbajele de programare din familia C.

2.4. Variabile locale

O variabilă locală este creată în interiorul unei funcții și nu poate fi accesată din afara ei. Funcții diferite pot avea variabile locale cu același nume pentru că ele nu se pot vedea una pe cealaltă. De fiecare dată când atribui o valoare unei variabile în interiorul unei funcții,

creezi o variabilă locală.

Să luam un exemplu de program (*pasari.py*)³⁰ care are două funcții și care fiecare conține o variabilă cu același nume:

```
#Acest program demonstreaza doua functii
#care au variabile locale cu acelasi nume.
def main():
    #Cheama functia dolj
    dolj()
    #Cheama functia gorj
    gorj()
#Defineste functia dolj. Ea creeaza
#o variabila locala numita pasari
def dolj():
    pasari = 5000
    print('Doljul are', pasari, 'de pasari')
#Defineste functia gorj. Ea creeaza
#o variabila locala numita pasari
def gorj():
    pasari = 8000
    print('Gorjul are', pasari, 'de pasari')
#Cheama functia principala
main()
```

Trecerea argumentelor la funcții

Un *argument* este o porțiune de date care este trecută într-o funcție atunci când funcția este invocată.

Un *parametru* este o variabilă care primește un argument ce este trecut într-o funcție.

Exemplu:

³⁰ Scrieți exemplele în editorul de text caracter cu caracter și nu le copiați (copy/paste) de pe formatul PDF în care se găsește cartea pentru că există mari șanse să primiți o eroare de tipul: "SyntaxError: invalid character in identifier". Aceasta se întâmplă pentru că formatele PDF, Word, ODT etc au caracteristici intime diferite, care cel mai adesea, nu se potrivesc interpretorului Python ce pur și simplu nu distinge ceea ce scrie în declarații.

```
def dublul (numar) :
    rezultat = numar*2
    print(rezultat)
```

Funcția se cheama dublul. Ea acceptă un număr (numar) ca parametru și afișează valoarea dublată a acelui număr. Sa vedem cum functioneaza in interpretorul Python:

```
>>> def dublul(numar):
...     numar=5
...     rezultat=numar*2
...     print(rezultat)
...
>>> dublul(5)
10
```

Fig.2.3. Exemplul anterior complet

Exemplu de program (*show_double.py*):

```
#Acest program demonstreaza cum un
#argument este trecut unei functii
def main() :
    valoare=5
    arata_dublul(valoare)
#Functia arata_dublul accepta un argument
#si afiseaza valoarea lui dubla
def arata_dublul (numar) :
    rezultat = numar*2
    print(rezultat)
#Cheama functia principala
main()
```

```
>>> def main():
...     valoare=5
...     arata_dublul(valoare)
...
>>> def arata_dublul(numar):
...     rezultat=numar*2
...     print(rezultat)
...
>>> main()
10
>>> █
```

Fig. 2.4. Trecerea unui argument la funcție

Trecerea într-o funcție a mai multor argumente

Există situații când trebuie să treci două sau mai multe argumente unei funcții.

```
def main():
    print('Suma lui 12 cu 45 este')
    arata_suma(12, 45)
def arata_suma(num1, num2):
    rezultat = num1 + num2
    print(rezultat)
#Cheama functia principala
main()
```

Să vedem exemplul în interpretor:

```
>>> def main():
...     print('Suma lui 12 cu 45 este')
...     arata_suma(12, 45)
...
>>> def arata_suma(num1, num2):
...     rezultat = num1+num2
...     print(rezultat)
...
>>> main()
Suma lui 12 cu 45 este
57
>>> █
```

Fig.2.5. Exemplul de mai sus în Python shell

Schimbarea parametrilor

```
def main():
    valoare = 99
    print('Valoarea este', valoare)
def schimba_valoarea(arg):
    print('Voi schimba valoarea.')
    arg = 0
    print('Acuma valoarea este', arg)
#Cheama functia principala
main()
```

2.5. Constante și variabile globale

O variabilă globală este accesibilă tuturor funcțiilor dintr-un program. Ea se creează în afara oricărei funcții.

Ex:

```
#Creeaza o variabila globala
numar = 0
def main():
    global numar
    numar = int(input('Introdu un numar '))
    arata_numar()
def arata_numar():
    print('Numarul introdus este: ', numar)
main()
```

Este totuși indicat să nu folosiți variabile globale. În schimb puteți să utilizați *constante globale*.

O constantă globală referă o valoare care NU POATE FI SCHIMBATĂ.

Prin convenție, constantele globale se scriu cu majuscule și sunt așezate la începutul programului, înaintea oricăror alte declarații, ca în exemplul următor:

```
>>> PI=3.14
>>> def aria_cercului(raza):
...     return PI*raza*raza
...
>>> def circumf_cerc(raza):
...     return 2*PI*raza
...
>>> print('Cercul cu raza 5 are suprafata: ', aria_cercului(5))
Cercul cu raza 5 are suprafata: 78.5
>>> print('Circumferinta aceluiași cerc este: ', circumf_cerc(5))
Circumferinta aceluiași cerc este: 31.400000000000002
>>>
>>> print('Circumferinta este: ', format(31.400000000, '.2f'))
Circumferinta este: 31.40
>>> █
```

Fig.2.6. Ilustrarea unei constate globale

2.6. Funcția lambda

Python permite crearea unei funcții anonime folosind cuvântul cheie `lambda`. O funcție anonimă poate să conțină doar o singură expresie care neapărat trebuie să returneze o valoare. Altfel ca la crearea unei funcții comune care folosește `def`, o funcție creată cu `lambda` returnează un *obiect funcție*. Acesta poate fi atribuit unei variabile care poate fi folosită în orice moment ca să execute expresia conținută. Să vedem cum arată în Terminal o funcție normală, aceeași funcție creată cu ajutorul cuvântului cheie `lambda` și o funcție `lambda` căruia nu-i atribuim nicio variabilă (da, permite și acest lucru!).

```
>>> def functie(x):  
...     return x*2  
...  
>>> functie(3)  
6  
>>>  
>>> functie=lambda x:x*2  
>>> functie(3)  
6  
>>>  
>>> (lambda x:x*2)(3)  
6  
>>> □
```

Fig.2.7. Funcția lambda

Capitolul III STRUCTURI DE DECIZIE

Una dintre cele mai importante, interesante și frumoase părți ale programării este oferită de structurile de decizie. Toți cei care au făcut programare în școală sau facultate – chiar dacă nu au fost deloc interesați de domeniu, le-a ajuns la ureche fie și total pasager cuvinte precum `if`, `while`, `else`, `continue`. Dacă ați terminat un liceu cu profil *real*, nu se poate să nu vă fi lovit de ele. Oricum ar fi, chiar dacă până acum nu știați de existența lor, a venit momentul să ne ocupăm de ele.

3.1. Declarația `if`

De cele mai multe ori într-un program ajungi la o răspântie, mai corect spus, în fața unei alegeri. Să ne imaginăm că mergem pe o șosea cu autoturismul personal și la un moment dat ajungem la o bifurcație. Să zicem de exemplu că ne deplasăm de la Craiova către București și am ajuns la Găneasa, Olt. Aici drumul se bifurcă: drumul spre dreapta duce la București via Pitești, drumul la stânga duce către Râmnicu Vâlcea, via Drăgășani. Să vedem cum arată drumul direct spre București „transpus” în Python cu ajutorul condiției `if` (daca):

```
if (daca) laDreapta:    #o iau la dreapta
    print('Ajung la Bucuresti prin Pitesti')
```

Dar poate că vrem să mergem la Râmnicu Vâlcea. Programul nostru ar arăta astfel în această variantă:

```
if(daca) laStanga:     #o iau la stanga
    print('Ajung la Ramnicu Valcea prin Dragasani')
```

Forma generală:

if *conditie*:

declaratie

declaratie

etc

Ex:

```
vanzari = float(input('Introdu vanzarile tale: '))  
if vanzari > 5000:  
    print('Ai un bonus de 500 de lei')
```

3.2. Operatori de comparatie

Nu putem sa mergem mai departe inainte de a vedea care sunt operatorii de comparatie in Python. Ei sunt prezentati mai jos:

Operator	Semnificație
<	Mai mic ca
>	Mai mare ca
<=	Mai mic sau egal cu
>=	Mai mare sau egal cu
==	Egal cu
!=	Diferit de (nu este egal cu)

Tabel 3.1. Operatori de comparație în Python

3.3. Declarația if-else

O declarație if-else execută un bloc de declarații dacă (if) condiția e adevărată sau alt bloc (else) dacă condiția e falsă.

Forma generală:

if conditie:

declaratie

declaratie

etc

else:

declaratie

declaratie

etc

Ex:

```
>>> temp = eval(input('Introdu temperatura:'))
Introdu temperatura:20
>>> if temp < 15:
...     print('Afara e cam racoare')
... else:
...     print('Afara e binisor')
...
Afara e binisor
>>> █
```

Fig.3.1. Ilustrarea clauzei if-else

Reguli de indentare a clauzei if-else

- fiti sigur că if și else sunt aliniate
- fiti sigur că declarațiile care urmează după if și else sunt indentate la rândul lor.

Compararea șirurilor

Python permite să compari șiruri cu ajutorul declarației if-else. Acest lucru îți dă voie să creezi structuri de decizie care testează valoarea unui șir.

Ex (*testare_nume.py*):

```
nume1 = 'Marcel'
nume2 = 'Marius'
if nume1 == nume2:
    print('Numele sunt la fel')
else:
    print('Numele sunt diferite')
```

Structuri de decizie imbricate³¹ și declarația if-elif-else

Ca să testeze mai mult de o condiție, o structură de decizie poate fi imbricată înăuntrul altei structuri de decizie.

Ex (*imprumut.py*):

```
MIN_SALARIU = 2000
MIN_ANI = 3
salariu = eval32(input('Introdu salariul tau: '))
ani_la_serviciu = eval(input('Introdu ani de serviciu:'))
```

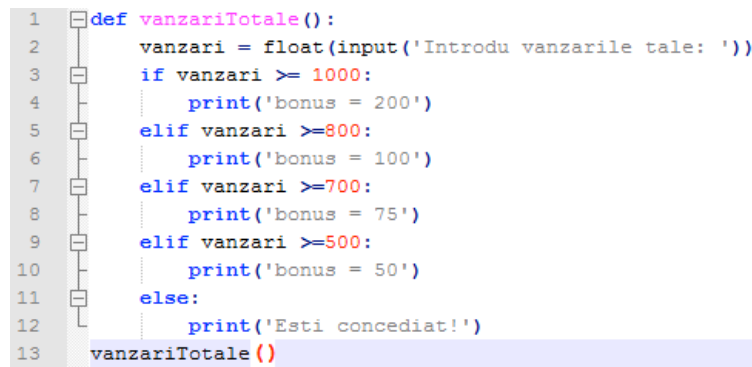
³¹ Imbricat = îmbrăcat(e), suprapus(e) ca șindrilele de pe casa (cf.dexonline.ro)

³² Atunci când introducem date numerice putem utiliza **eval** fără să ne mai gândim dacă e vorba de o valoare în virgulă mobilă sau de un întreg. **eval** ține loc de float sau input.

```

if salariu >= MIN_SALARIU:
    if ani_la_serviciu >= MIN_ANI:
        print('Te califici pentru imprumut')
    else:
        print('Nu te califici')
else:
    print('Trebuie sa ai minim..', MIN_ANI)

```



```

1 def vanzariTotale():
2     vanzari = float(input('Introdu vanzarile tale: '))
3     if vanzari >= 1000:
4         print('bonus = 200')
5     elif vanzari >= 800:
6         print('bonus = 100')
7     elif vanzari >= 700:
8         print('bonus = 75')
9     elif vanzari >= 500:
10        print('bonus = 50')
11    else:
12        print('Esti concediat!')
13    vanzariTotale()

```

Fig.3.2. Declarația if-elif-else

Pentru a ușura și a face economie de efort și de spațiu se poate utiliza declarația (clauza) if-elif-else. Mai sus este un exemplu de program ce include declarația if-elif-else; În momentul în care userul introduce diverse valori cuprinse între 500 și 1000, programul va afișa bonificația corespunzătoare. Dacă însă introduce o valoare mai mică de 500, programul afișează șirul „Ești concediat!”.

3.4. Operatori logici

Sunt: *and*, *or*, *not*.

Exemplu *and*:

```

if temperatura < 20 and minute > 12:
    print('Temperatura e periculoasa')

```

Exemplu *or*:

```

if temperatura < 20 or temperatura > 100:
    print('Temperatura este extrema')

```

Exemplu *not*:

```

if not(temperatura > 100):

```

```
print('Aceasta e aproape temperatura maxima')
```

3.5. Variabile booleene

O variabilă *booleană* poate referi două valori: `TRUE` sau `FALSE`. Ele arată dacă o condiție există. Variabilele booleene sunt folosite ca indicatori. Un *indicator* este o variabilă care semnalizează când o condiție există în program.

Ex:

```
if cota_vanzari >=5000.0:
    cota_vanzari_ok = True
else:
    cota_vanzari_not_ok = False
```

3.6. Bucle

Buclele sunt acele părți dintr-un program care sunt folosite ca să execute o declarație atâta timp cât o expresie este adevărată.

3.7. Bucla `while` – o buclă controlată

O condiție controlată face ca o declarație sau un bloc de declarații să se repete atâta timp cât o condiție e adevărată. Python folosește declarația (bucla) **`while`** ca să scrie o astfel de condiție.

Formatul general al buclei *while* este:

while *conditie*:

declaratie

declaratie

etc

Ex:

```
while valoare == 'y':
```

Exemplu:

```

>>> numara=0
>>> while(numara<9):
...     print('Numaratoarea este: ', numara)
...     numara=numara+1
...     print('La revedere!')
...
Numaratoarea este: 0
La revedere!
Numaratoarea este: 1
La revedere!
Numaratoarea este: 2
La revedere!
Numaratoarea este: 3
La revedere!
Numaratoarea este: 4
La revedere!
Numaratoarea este: 5
La revedere!
Numaratoarea este: 6
La revedere!
Numaratoarea este: 7
La revedere!
Numaratoarea este: 8
La revedere!
>>> █

```

Fig. 3.3. Bucla while

Bucla `while` nu se execută niciodată dacă condiția de început e falsă.

Să mai luăm un exemplu de buclă `while`. Vom scrie un program care transformă gradele Fahrenheit în Celsius.

Formula de transformare este:

$$C = 5/9 * (F-32)$$

(*temp.py*)

```

>>> temp=0
>>> while temp != -100:
...     temp = eval(input('Introdu temp. in F (-100 ca sa iesi):'))
...     print('Temp. in C este: ', format(5/9*(temp-32), ',.2f'))
...
Introdu temp. in F (-100 ca sa iesi):90
Temp. in C este: 32.22
Introdu temp. in F (-100 ca sa iesi):67
Temp. in C este: 19.44
Introdu temp. in F (-100 ca sa iesi):89
Temp. in C este: 31.67
Introdu temp. in F (-100 ca sa iesi):-100
Temp. in C este: -73.33
>>> █

```

Fig.3.4. Bucla while

Este nevoie ca la începutul programului să setăm `temp = 0` pentru că altfel primim o eroare. Programul pornește, iar bucla `while` caută să vadă dacă variabila `temp` nu cumva este egală cu `-100`. Dacă `temp` nu a fost setată inițial cu `0`, se creează o problemă pentru că `temp` nu există și bucla `while` nu știe ce să facă! În locul valorii `0` putem pune oricare altă cifră dar nu `-100`. Dacă facem asta, bucla `while` devine falsă dintr-un început și nu rulează niciodată.

O buclă `while` e asemănătoare condiționalei `if` cu diferența că cea din urmă funcționează

doar o singură dată iar while până când condiția ei este atinsă (în cazul nostru `temp = -100`).

Să șlefuiim puțin exemplul de mai sus, astfel încât atunci când utilizatorul introduce valoarea -100, computerul să-i spună „La revedere!”:

```
temp = 0
while temp != -100:
    temp=eval(input('Introdu temp. in F(-100 ca sa iesi):' ))
    if temp != -100:
        print('Temp in C este: ', format(5/9*(temp-32), '.2f'))
    else:
        print('La revedere!')
```

3.8. Bucla infinită

Să spunem că avem următorul exemplu de buclă while:

```
numar = 0
while numar < 10:
    print(numar)
```

În acest program valoarea variabilei `numar` nu se schimbă niciodată și deci declarația `numar < 10` este întotdeauna adevărată. Python va afișa la infinit numărul 0. Ca să oprim bucla infinită trebuie pur și simplu să întrerupem brutal shell-ul Python. Există însă o posibilitate de a întrerupe o buclă infinită atunci când scriem programul. Pentru aceasta folosim declarația `break`:

```
x = 0
numar = 1
while x < 10 and numar > 0:
    numar = int(input('Introdu un numar: '))
    break
```

3.9. Bucla for

Bucla for iterează de un număr specific de ori.

Format general:

for variable in [value1, value2, etc] :

declaratie

declaratie

etc

Ex:

```
#Acest program demonstreaza o bucla for
#care utilizeaza o lista de numere
def main():
    print('Voi afisa numerele de la 1 la 5')
    for num in [1, 2, 3, 4, 5]:
        print(num)
#Cheama functia principala
main()
```

```
>>> def main():
...     print('Voi afisa numerele de la 1 la 5')
...     for num in [1,2,3,4,5]:
...         print(num)
...
>>> main()
Voi afisa numerele de la 1 la 5
1
2
3
4
5
```

Fig.3..5. Bucla for

Folosirea funcției range cu bucla for

Funcția range creează un tip de obiect numit **iterabil**. Un *iterabil* este similar unei *liste* (vom vorbi ceva mai târziu despre liste):

```
for num in range(5):
    print (num)
```

```
>>> for num in range(5):
...     print(num)
...
0
1
2
3
4
>>> █
```

Fig.3.6. Bucla *for...in range*

Ex:

```
#Acest program demonstreaza cum functia range
#poate fi folosita cu bucla for
```

```
def main():
    #Afiseaza un mesaj de 5 ori
    for x in range(5):
        print('Ce faci Caine?!')
#Cheama functia principala
main()

>>> def main():
...     for x in range(5):
...         print('Ce faci Caine?!')
...
>>> main()
Ce faci Caine?!
Ce faci Caine?!
Ce faci Caine?!
Ce faci Caine?!
Ce faci Caine?!
>>> █
```

Fig.3.7. Exemplu de mai sus în interpretor

Funcția `for ...in range` produce o secvență de numere care crește cu valoarea 1 fiecare număr succesiv din listă până când este atinsă valoarea 5 inclusiv. Să vedem ce se întâmplă cu următoarea declarație:

```
for numar in range(1, 10, 2):
    print(numar)
```

Primul argument este 1 iar ultimul argument este 10. Dar ce reprezintă numărul 2 ? Acesta este folosit ca **valoarea de pas** (*step valor sau pas*). Astfel, fiecare număr succesiv din secvență în loc să crească cu 1, va crește cu valoarea pasului, în cazul nostru 2. Deci 2 va fi adăugat fiecărui număr succesiv din secvență:

```
>>> for num in range(1, 10, 2):
...     print(num)
...
1
3
5
7
9
>>> █
```

Fig.3.8. Demonstrarea valorii de pas

Să vedem ce se întâmplă dacă o luăm de la sfârșit spre început iar valoarea pasului devine negativă:

```

>>> for num in range(10, 1, -2):
...     print(num)
...
10
8
6
4
2
>>> █

```

Fig.3.9. Inversarea secvenței

3.10. Acumulatori

Un *total de funcționare* (running total) este o sumă de numere care acumulează fiecare iterare (trecere, execuție) a unei bucle. Variabila folosită ca să înmagazineze totalul se cheamă **acumulator**.

Multe programe cer să calculezi totalul unei serii de numere. De exemplu, se presupune că scrii un program care calculează totalul vânzărilor pentru o săptămână. Programul citește vânzările din fiecare zi și calculează totalul acelor numere.

Acest tip de programe folosește de obicei două elemente:

1. o buclă care citește fiecare număr al seriei
2. o variabilă care acumulează totalul numerelor citite

Să luăm un program care demonstrează un acumulator:

#Acest program calculeaza suma unei serii


#de numere introduse de user

#Ia constanta pentru numarul maxim

MAX = 5  (Reține! Constanta se pune la început)

Def main():

 #Initializeaza variabila acumulator

total = 0.0  (acumulatorul se inițializează întotdeauna cu 0.0)

 #Explica ceea ce faci

 print('Acest program calculeaza suma')

 print(MAX, 'numerelor introduse.')

 #Ia numerele si acumuleaza-le

 for counter in range(MAX):

 number = int(input('Introdu un numar: '))

```

        total = total+number
    #Afiseaza totalul numerelor
    print('Totalul este', total)
#Cheama functia principala
main()

```

3.11. Operatori de atribuire augmentată

Aceștia ajută la prescurtarea și deci simplificarea codului.

Exemple:

`x = x + 1` se mai poate scrie `x+=1`

sau:

`y = y - 2` se mai poate scrie `y-=2`

sau:

`z = z * 5` se mai poate scrie `z*=5`

sau

`total = total + number` devine `total += number`

3.12. Bucle de validare a intrărilor

Validarea intrărilor (input-ului) este procesul de verificare a datelor care sunt introduse într-un program ca să fii sigur că sunt corecte înainte de a fi folosite în calcule. Dacă un user introduce date greșite, programul va scoate răspunsuri eronate (sau erori logice care sunt cel mai greu de depanat...). Calculatorul – oricât de deștept l-am crede (și nu este...) – procesează datele pe care *omul* i le furnizează. El nu știe să facă diferența între datele bune și cele proaste.

De exemplu, următorul program va afișa un rezultat eronat pentru că datele introduse sunt ilogice (*salariat.py*):

```

#Acest program afiseaza salariul unui angajat
def main():
    #Ia numarul orelor lucrate intr-o saptamana
    ore = int(input('Introdu numarul orelor lucrate intr-o saptamana:
'))
    #Ia salariul orar

```

```

sal_orar=float(input('Introdu salariul orar: '))
#Calculeaza salariul
salariu = ore * sal_orar
#Afiseaza salariul
print('Salariul este: lei ', format(salariu, ',.2f'))
#Cheama functia main
main()

```

```

192-168-0-100:desktop mirceaprodan$ python salariat.py
Introdu numarul orelor lucrate saptamanal: 400
Introdu salariul pe ora: 8.00
Salariu este: lei 3,200.00
192-168-0-100:desktop mirceaprodan$ █

```

Fig. 3.10. Date aberante și răspunsuri pe măsură!

Ce s-ar întâmpla dacă la orele lucrate pe săptămână, un angajat ar introduce în loc de 40 de ore (adică opt ore pe zi înmulțite cu 5 zile pe săptămână) 400 de ore? Rezultatul ar fi aberant, în primul rând pentru că o săptămână – fie ea de lucru sau nu – nu poate să aibă 400 de ore!

Cum corectăm o asemenea năzbâtie? Simplu! Adăugăm o buclă while:

```

#Acest program afiseaza salariul unui angajat
def main():
    #Ia numarul orelor lucrate intr-o saptamana
    ore = int(input('Introdu nr orelor lucrate intr-o saptamana: '))
    #Asigura-te ca userul nu introduce o valoare absurda
    while ore > 40:
        print('Eroare! Nu poti lucra mai mult de 40 de ore!')
        ore=int(input('Introdu orele corecte: '))
    #Ia salariul orar
    sal_orar=float(input('Introdu salariul orar: '))
    #Calculeaza salariul
    salariu = ore * sal_orar
    #Afiseaza salariul
    print('Salariul este: ', format(salariu, ',.2f'))
#Cheama functia main
main()

```

3.13. Bucle imbricate

O buclă aflată înăuntrul altei bucle se cheamă imbricată.

Ceasul este cel mai bun exemplu de bucle imbricate. Secunde, minutele și orele se rotesc pe un ecran de ceas. O oră se rotește complet în 12 “ore” ale ecranului. Unui minut îi trebuiesc 60 de rotiri. Pentru a demonstra “funcționarea” ceasului folosim o buclă `for`:

```
for secunde in range(60):  
    print (secunde)
```

sau

```
for minute in range(60):  
    for secunde in range(60):  
        print(minute, ':', secunde)
```

Buclele complete ale ceasului arată așa:

```
for ore in range(24):  
    for minute in range(60):  
        for secunde in range(60):  
            print(ore, ':', minute, ':', secunde)
```

Bucla cea mai din interior iterează de 60 de ori pentru fiecare iterație a buclei din mijloc.

Bucla din mijloc iterează de 60 de ori la fiecare iterație a buclei din exterior. Când bucla din exterior iterează de 24 de ori, bucla din mijloc iterează de 1440 de ori iar cea din interior de 86.400 de ori.

Capitolul IV Module

O funcție care returnează valori este o funcție care *trimite înapoi* o valoare părții din program care a chemat-o. Python oferă o bibliotecă (pre) scrisă de funcții care face asta. Această bibliotecă conține o funcție care generează numere aleatoare (întâmplătoare).

4.1. Biblioteci de funcții standard și declarația import

Am folosit deja câteva funcții standard pre-scrise în biblioteci: `print`, `input`, `range`, `type`.

Câteva dintre funcțiile Python sunt construite în interpretorul lui. Dacă vrei să le folosești, pur și simplu le invoci. Așa este cazul cu **`input`**, **`print`**, **`range`**. Multe alte funcții sunt însă stocate în biblioteci numite **module**. Aceste module - care sunt copiate în computer în momentul instalării Python, ajută la organizarea bibliotecilor standard.

Când invoci o funcție stocată într-un modul, trebuie să scrii o declarație de import în partea cea mai de sus a programului. Spre exemplu, să luăm modulul numit `math`. El conține funcții matematice care lucrează cu numere reale (în virgulă mobilă). Dacă vrei să folosești modulul `math` trebuie să scrii în vârful programului o declarație ca aceasta:

```
import math
```

Această declarație face ca interpretorul să încarce conținutul modulului `math` în memorie și să-l facă disponibil.

4.2. Generarea numerelor aleatoare

Python oferă câteva biblioteci de funcții care lucrează cu numere aleatoare (întâmplătoare). Acestea sunt stocate într-un modul numit `random`. Ca să generăm numere aleatoare vom importa modulul `random` prin declarația:

```
import random
```

Prima funcție aleatoare generată se numește `randint`. De fapt există forme mai explicite de a importa modulele și funcțiile din el:

```
from random import randint
```



```
from random import *
```

Pentru că funcția *randint* se găsește în modulul *random*, avem nevoie să folosim **notația cu punct** ca să ne referim la ea.

Iată mai jos cum se scrie o astfel de declarație:

modul
random.randint ← funcție

În partea stângă a punctului este numele modulului (*random*) iar după el numele funcției (*randint*).

Deci, reține că:

1. Punctul leagă modulul de funcție.
2. *random* este modulul.
3. *randint* este funcția.

Ex:

```
number = random.randint(1, 100)
```

Argumentul (1, 100) spune funcției *randint* să afișeze un număr întreg aleator situat între 1 și 100.

Exemplu de program:

```
#Acest program afiseaza un numar aleator
#situat in marja 1 pana la 100
import random
def main():
    #Ia un numar aleator
    numar = random.randint(1, 100)
    #Afiseaza numarul
    print('Numarul este', numar)
#Cheama functia principala
main()
```

```
>>> #Acest program afiseaza un numar aleator
.. #situat in marja 1 la 100
.. import random
>>> def main():
..     #Ia un numar aleator
..     numar=random.randint(1, 100)
..     #Afiseaza numarul
..     print('Numarul este: ', numar)
>>> #Cheama functia main
.. main()
.. Numarul este: 85
>>>
```

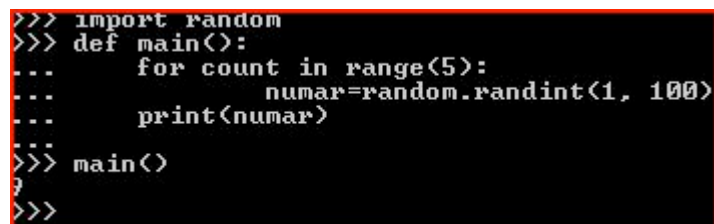
Fig.4 .1. Numere aleatoare

Acuma să luăm un alt exemplu în care iterăm cu o buclă *for* de 5 ori:

```
#Acest program afiseaza 5 numere aleatoare
#situate intre 1 si 100
import random
def main():
    for count in range(5):
        #Ia un numar aleator
        numar = random.randint(1, 100)
        #Afiseaza numarul
        print(numar)
#Cheama main
main()
```

Sa simplificăm programul anterior astfel:

```
import random
def main():
    for count in range(5):
        print(random.randint(1, 100))
main()
```



```
>>> import random
>>> def main():
...     for count in range(5):
...         numar=random.randint(1, 100)
...         print(numar)
>>> main()
?
>>>
```

Fig.4.2. Ilustrarea exemplului de mai sus

4.3. Funcțiile *randrange*, *random* și *uniform*

Funcția *randrange* ia același argument ca funcția *range*. Diferența este că *randrange* nu returnează o listă de valori. În loc de asta, ea returnează o valoare aleatoare dintr-o secvență de valori.

De exemplu, următoarea declarație atribuie un număr aleator situat între 0 și 9 variabilei *numar*:

```
numar = random.randrange(10)
```

Argumentul – în cazul nostru 10 – specifică limita unei secvențe de valori. Funcția va returna un număr aleator selectat din secvența de la 0 în sus dar nu include limita

sfârșitului, adică numărul 10.

Următoarea declarație specifică și valoarea de început dar și de sfârșit a secvenței:

```
numar = random.randrange(5, 10)
```

Când această declarație e executată, un număr întâmplător cuprins între 5 și 9 va fi atribuit variabilei `numar`.

Următorul exemplu specifică o valoare de start, una de sfârșit și o altă valoare:

```
numar = random.randrange(0, 101, 10)
```

Funcția uniform returnează un număr aleator în virgulă mobilă, dar îți permite să specifici media valorilor pe care le-ai selectat:

```
numar = random.uniform(1.0, 10.0)
```

Declarația de mai sus face ca funcția *uniform* să returneze o valoare aleatoare în virgulă mobilă situată în gama 1.0 până la 10.0 și s-o atribuie variabilei `numar`.

4.4. Scrierea propriei funcții care returnează o valoare

O funcție care returnează o valoare conține o declarație `return` care înapoiază o valoare părții de program care a invocat-o.

Scrierea unei funcții care returnează o valoare se face la fel ca scrierea unei funcții simple cu o excepție: o funcție care returnează o valoare trebuie să aibă o declarație `return`.

Forma generală a funcției este:

```
def function_name():  
    declaratie  
    declaratie  
    etc  
    return expression
```

Ex:

```
def sum(num1, num2):  
    result = num1 + num2  
    return result
```

Și acum un exemplu practic (*total_bani.py*):

```
#Acest program foloseste valoarea returnata a unei functii  
def main():  
    #Ia valoarea banilor unui cetatean
```

```

bani_primul=float(input('Introdu valoarea banilor: '))
#Ia valoarea banilor celui de-al doilea cetatean
bani_al_doilea=float(input('Introdu valoarea banilor: '))
#Afla valoarea totala
total=sum(bani_primul, bani_al_doilea)
#Afiseaza totalul
print('Impreuna cei doi au: ' total)
#Functia sum accepta doua argumente numerice si
#returneaza suma celor doua argumente
def sum(num1, num2):
    rezultat=num1 + num2
    return rezultat
#Cheama main
main()

```

4.5. Modularea cu funcții

(exemplu de aplicație)

Andrei are o afacere numită “Fă-ți propria muzică” prin intermediul căreia vinde instrumente muzicale. Andrei își plătește angajații cu comision. Comisionul este în funcție de vânzări astfel:

- mai puțin de 10000 lei - 10%
- 10000 – 14999 – 12%
- 15.000 – 17.999 – 14%
- 18.000 – 21.999 – 16%
- 22.000 și mai mult – 18%

Pentru că agenții de vânzări sunt plătiți lunar, Andrei permite fiecăruia dintre ei să ia în avans câte 2000 de lei. Când comisionul este calculat, suma pe care fiecare angajat a luat-o în avans este scăzută din comision. Dacă comisionul este mai mic decât suma luată în avans, ei trebuie să-i ramburseze lui Andrei diferența. Ca să calculeze plata lunară a fiecărui angajat, Andrei folosește următoarea formulă:

$$plata = vanzari * comision - avans$$

Andrei te roagă să-i scrii o aplicație care să facă toate calculele pentru el.

Algoritmul aplicației este următorul:

1. Ia vânzările lunare ale fiecărui agent de vânzări
2. Ia suma de bani luată în avans de fiecare dintre ei
3. Folosește valoarea vânzărilor lunare ca să afli comisionul
4. Calculează plata fiecărui angajat folosind formula de mai sus. Dacă valoarea e negativă, angajatul trebuie să restituie banii.

Programul (*comision.py*):

```
#Acest program calculeaza salariul
#unei persoane de vanzari
def main():
    #Ia suma vanzarilor
    vanzari = ia_vanzarile()
    #Ia valoarea luata in avans
    avans = ia_avansul()
    #Determina comisionul
    comision = determina_comision(vanzari)
    #Calculeaza plata
    plata = vanzari*comision - avans
    #Afiseaza valoarea platii
    print ('Plata este lei', format(plata, '.2f'))
    #Afla daca plata e negativa
    if plata < 0:
        print('Plata trebuie rambursata')
    else:
        print('Plata nu trebuie rambursata')
#Cheama functia principala
main()
```

4.6. Modulul matematic

Modulul *math* conține numeroase funcții care pot fi folosite în calcule matematice

Ex:

```
rezultat = math.sqrt(16)
```

Funcția `sqrt` acceptă argumentul 16 și îi returnează rădăcina pătrată (care e 4).

Mai întâi trebuie să importăm modulul `math` pentru a scrie un program care îl folosește.

Ex de program (*radacina.py*):

```
#Acest program demonstreaza functia sqrt
import math
def main():
    #Ia un numar
    numar = float(input('Introdu un numar: '))
    #Ia radacina patrata a numarului
    radacina_patrata = math.sqrt(numar)
    #Afiseaza radacina patrata a numarului
    print('Radacina patrata este', radacina_patrata)
#Cheama functia principala
main()
```

Următorul program folosește funcția `hypot` ca să calculeze ipotenuza unui triunghi dreptunghic:

```
#Acest program calculeaza lungimea ipotenuzei
#unui triunghi dreptunghic
import math
def main():
    #Ia lungimea a doua laturi ale triunghiului
    a=float(input('Introdu lungimea laturii A: '))
    b=float(input('Introdu lungimea laturii B: '))
    #Calculeaza lungimea ipotenuzei
    c = math.hypot(a, b)
    #Afiseaza lungimea ipotenuzei
    print('Lungimea ipotenuzei este: ', c)
#Cheama functia main
main()
```

Valorarea `math.pi`

Suprafața cercului este constanta π înmulțită cu raza cercului la pătrat, după cum bine știm din clasa a VII-a, de la studiul geometriei plane:

```
suprafata = math.pi*radius**2
```

Ce se întâmplă? Suprafața cercului este $S=\pi \cdot r^2$ (unde r este raza cercului) la pătrat. Dar cum π este o constantă universală care aparține modulului matematic, trebuie să scriem `math.pi`.

Un modul este doar un fișier care conține cod Python. Programele mari sunt (mai) ușor de reparat și întreținut atunci când sunt împărțite în module dar mai ales pot fi refolosite³³ într-un alt program. Un modul poartă întotdeauna extensia `.py`.

Modulul calendar

Sa luam de exemplu modulul implicit Python `calendar`. El se importa ca oricare alt modul, asa cum am aratat mai devreme, cu declaratia `import`:

```
import calendar
```

Ca să afișăm de exemplu cum arăta anul Revoluției 1989 trebuie să ne referim la funcția inclusă în modulul `calendar`, `prcal` (*pr*intează *calendar*) folosind notația cu punct:

```
calendar.prcal()
```

Programul următor afișează anul amintit:

```
import calendar
an = int(input('Introdu anul dorit: '))
prcal(an)
```

Să vedem cum funcționează în Python shell:

```
>>> import calendar
>>> an = int(input('Introdu anul dorit: '))
Introdu anul dorit: 1989
>>> calendar.prcal(1989)
```

1989

January								February								March							
Mo	Tu	We	Th	Fr	Sa	Su		Mo	Tu	We	Th	Fr	Sa	Su		Mo	Tu	We	Th	Fr	Sa	Su	
						1				1	2	3	4	5				1	2	3	4	5	
2	3	4	5	6	7	8		6	7	8	9	10	11	12		6	7	8	9	10	11	12	
9	10	11	12	13	14	15		13	14	15	16	17	18	19		13	14	15	16	17	18	19	
16	17	18	19	20	21	22		20	21	22	23	24	25	26		20	21	22	23	24	25	26	
23	24	25	26	27	28	29		27	28							27	28	29	30	31			
30	31																						

April								May								June							
Mo	Tu	We	Th	Fr	Sa	Su		Mo	Tu	We	Th	Fr	Sa	Su		Mo	Tu	We	Th	Fr	Sa	Su	
					1	2		1	2	3	4	5	6	7					1	2	3	4	
3	4	5	6	7	8	9		8	9	10	11	12	13	14		5	6	7	8	9	10	11	
10	11	12	13	14	15	16		15	16	17	18	19	20	21		12	13	14	15	16	17	18	

Fig.4.3. Modulul calendar

³³ Reuse în lib. engleză

Capitolul V Fișiere și excepții

Când un program are nevoie să salveze date pentru a le folosi mai târziu, el le scrie într-un **fișier**. Datele pot fi extrase și citite din fișier în orice moment.

Programele pe care le scrii sunt refolosite pentru că datele lui sunt stocate în memoria RAM (Random Acces Memory). Datele sunt salvate într-un fișier care este stocat pe disc. Odată ce datele sunt salvate în fișier, ele rămân și după ce fișierul este închis. Datele pot fi retrimise și refolosite de utilizator oricând dorește.

Exemple:

- procesoare word
- editoare de imagine
- tabele de date
- jocuri de calculator
- browsere web (prin cookie)

Când o bucatică de date este scrisă într-un fișier, ea e copiată dintr-o variabilă din RAM în fișierul de pe disc.

Procesul de retrimiteră a datelor dintr-un fișier este cunoscut ca *citirea din fișiere*.

Procesul acesta este invers scrierii în fișiere: când o porțiune de date este citită din fișierul de pe HDD, este copiată în RAM și referită de o variabilă.

Procesul implică trei pași:

1. Deschiderea fișierului – creează o conexiune între fișier și program. Deschiderea unui fișier din input permite programului să citească date din el.
2. Procesarea fișierului – datele sunt scrise în fișier (fișier output) sau citite (fișier input)
3. Închiderea fișierului – când programul termină cu fișierul, el trebuie închis.
- 4.

5.1. Tipuri de fișiere

Sunt două tipuri de fișiere: **text** și **binare**.

Un fișier text conține date care au fost codificate în text folosind scheme ca ASCII sau Unicode. Chiar dacă fișierele conțin numere, ele sunt stocate ca o serie de caractere. Drept rezultat, fișierul poate fi deschis și văzut într-un editor de text precum Notepad sau Word. Un fișier binar conține date care sunt convertite în text. Ca urmare, nu le poți citi cu un editor de text.

5.2. Metode de acces a fișierelor

Majoritatea limbajelor de programare oferă două căi de acces la datele stocate în fișiere:

- a) acces secvențial
- b) acces direct

Când lucrezi cu accesul secvențial, accesezi date de la început spre sfârșit. Deci dacă vrei să citești date situate spre sfârșitul fișierului ești nevoit să parcurgi **tot** fișierul – **nu poți sări** la ceea ce te interesează. *Este ca la casetofon*: nu poți pune direct cântecul dorit.

Când însă folosești accesul direct (random access file), poți sări direct la orice date din fișier. *Este ca la CD sau ca la pick-up*: poți pune direct orice cântec vrei.

Noi vom lucra cu accesul secvențial.

Numele fișierului și obiectele fișier (Filenames and File Objects)

Fișierele sunt identificate printr-un nume. Asta se întâmplă când le salvăm. De exemplu: *cat.jpg ; nota.txt ; scrisoare.doc* etc.

În măsura în care un program lucrează cu un fișier de pe computer, programul trebuie să creeze un fișier obiect în memorie ca să-l poată accesa. Un **fișier obiect** (*file object*) este un obiect asociat cu un fișier specific și oferă o cale programului de a lucra cu acel fișier.

În program, o variabilă referă obiectul fișier. Acea variabilă se îngrijește de toate operațiile făcute în fișier.

5.3. Deschiderea unui fișier în Python

Ca să deschidem un fișier folosim funcția **open**. Funcția `open` creează un fișier obiect pe care îl asociază cu fișierul dorit de pe discul computerului:

Formatul general al funcției `open` este:

```
file_variable = open(filename, mode)
```

unde:

- *file_variable* este numele variabilei care referă obiectul fișier
- *filename* este un șir care arată numele fișierului
- *mode* este un șir care specifică *modul* (scris, citit, etc) în care fișierul va fi deschis.

Exemple de moduri (mode) în Python:

'r' = deschide un fișier doar pentru citit (read); fișierul de acest tip nu poate fi scris

'w' = deschide un fișier pentru scris (write). El poate fi interpretat cam așa: dacă fișierul există, șterge-i conținutul; dacă nu există, creează-l.

'a' = deschide un fișier pentru a fi scris. Toate datele fișierului vor fi anexate până la sfârșit.

Dacă nu există (fișierul) creează-l.

De exemplu, să presupunem că fișierul *client.txt* conține datele unui client și vrem să-l deschidem pentru a-l citi. Iată un exemplu de cum invocăm funcția open:

```
fișier_client = open('client.txt', 'r')
```

După ce declarația este executată, fișierul *client.txt* este deschis și variabila *fișier_client* va referi un fișier obiect pe care-l putem folosi ca să citim date din fișier.

Acuma, vrem să creăm un fișier numit *vanzari.txt* și să scriem în el. Iată cum facem:

```
fișier_vanzari = open('vanzari.txt', 'w')
```

5.4. Scrierea datelor într-un fișier

Este timpul să introducem un alt tip de funcții care se cheamă *metode*.

O **metodă** este o funcție care aparține unui obiect și care face unele operații folosind acel obiect. Odată ce ai deschis un fișier, folosești metoda fișierului obiect ca să poți face operații pe fișier.

De exemplu, obiectul fișier are o metodă numită **'write'** care poate fi folosit ca să scrii date într-un fișier.

Iată formatul general și cum să invoci metoda *write*:

```
file_variable.write(string)
```

În această declarație, *file_variable* este o variabilă care referă un fișier obiect și *string* este un șir care va fi scris în fișier. Fișierul trebuie să fie deschis pentru scris ('w' sau 'a') altfel va

apărea o eroare.

Să spunem că *customer_file* referă un fișier obiect și fișierul va fi deschis pentru scris cu modul 'w'. Iată cum vom scrie de exemplu șirul 'Traian Basescu' într-un fișier:

```
customer_file.write('Traian Basescu')
```

Următorul fragment de cod arată un alt exemplu:

```
nume = 'Traian Basescu'
customer_file.write('nume')
```

Odată ce am terminat cu un fișier, el trebuie închis cu cuvântul **close**. Ex.:

```
customer_file.close()
```

Exemplu complet de program (*presedinti.py*)

```
#Acest program scrie trei linii de date
#intr-un fisier
def main():
    #Deschide un fisier numit presedinti.txt
    outfile = open('presedinti.txt', 'w')           5
    #Scrie numele presedintilor Romaniei           6
    #in fisier
    outfile.write('Ion Iliescu\n')                  8
    outfile.write('Emil Constantinescu\n')          9
    outfile.write('Traian Basescu\n')              10
    #Inchide fisierul
    outfile.close()                                 12
#Cheama functia principala
main()
```

Rezultatul programului este redat mai jos:

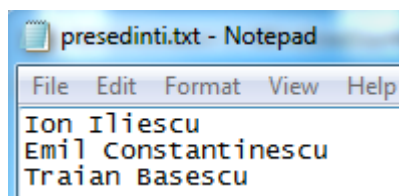


Fig.5.1 *presedinti.txt*

(**Nota:** Numerele din partea dreaptă reprezintă rândurile programului). Cum lucrează însă acest program? Linia 5 deschide fișierul *tenori.txt* folosind modul 'w'. Acesta face ca fișierul să fie creat și bun pentru scris în el. De asemenea, aceeași declarație creează un obiect în

memorie pe care îl atribuie variabilei `outfile`. Declarațiile din liniile 8, 9 și 10 scriu pur și simplu șirurile din fișier. Linia 12 închide fișierul. După ce programul este rulat, numele celor trei personaje este scris în fișierul *tenori.txt*. Să mai observăm că fiecare șir se termină cu `\n` adică numele următorului tenor este așezat pe o linie nouă.

5.5. Citirea datelor dintr-un fișier

Dacă un fișier este deschis pentru citire (folosind modul `'r'`), putem folosi metoda obiect **read** ca să-i citim întregul conținut. Când este invocată metoda *read*, ea returnează conținutul fișierului ca pe un șir. Să vedem în exemplul următor cum folosim metoda *read* ca să citim conținutul textului *tenori.txt* pe care l-am creat anterior:

(*citire_fisier.py*)

```
#Acest program citește și afișează conținutul
#fișierului presedinti.txt

def main():

    #deschidem fișierul numit presedinti.txt

    infile = open('presedinti.txt', 'r')           5

    #Îi citim conținutul

    file_contents = infile.read()                 7

    #Închidem conținutul

    infile.close()

    #Afișăm datele citite

    print(file_content)

#Invocăm funcția principală

main()
```

Declarația din linia 5 deschide fișierul pentru citire folosind modul `'r'`. De asemenea creează un fișier obiect pe care îl atribuie variabilei `infile`. Linia 7 invocă metoda `infile.read` ca

să îi citească conținutul. Conținutul este citit în memorie ca un șir și atribuit variabilei `file_contents`. Putem de asemenea folosi metoda `readline` ca să citim doar o linie dintr-un fișier. Metoda returnează o linie ca pe un șir.

Concatenarea unei linii noi la un șir

În cele mai multe cazuri, datele care sunt scrise într-un fișier nu sunt șiruri literare dar sunt referite în memorie de variabile. Este cazul în care un program invită userul să introducă date și apoi să le scrie într-un fișier. Când un program scrie date introduse de user, este de obicei necesar să le legăm (concatenăm) cu un caracter `\n`. Acest lucru ne va asigura că fiecare dată este scrisă într-o linie nouă în fișier. Programul următor ne arată cum se face acest lucru (*barbati.py*).

```
#Acest program ia trei nume de la user
#Si le scrie intr-un fisier
def main():
    print('Introdu numele a trei barbati.')
    nume1 = input('Barbatul #1 ')
    nume2 = input('Barbatul #2 ')
    nume3 = input('Barbatul #3 ')
    #deschide un fisier numit barbati.txt
    fisier = open('barbati.txt', 'w')
    #Scrie numele in fisier
    fisier.write(nume1 + '\n')
    fisier.write(nume2 + '\n')
    fisier.write(nume3 + '\n')
    #inchide fisierul
    fisier.close()
    print('Numele au fost scrise in barbati.txt')
main()
```

Sa verificam existenta fisierului si functionalitatea lui:

```
192-168-0-100:desktop mirceaprodan$ python3 barbati.py
Introdu numele a trei barbati.
Barbatul #1 Costel
Barbatul #2 Mirel
Barbatul #3 Florel
Numele au fost scrise in barbati.txt
192-168-0-100:desktop mirceaprodan$
```

Fig.5.2. Deschidere fisier

Ca sa rulam programul *barbati.py* trebuie neaparat sa fim in acest director. Tot aici va fi creat si salvat fisierul *barbati.txt*.

5.6. Adăugarea datelor într-un fișier existent

Când folosim metoda 'w' ca să deschidem un fișier dar fișierul cu nume specificat există deja pe disc, fișierul existent va fi șters și unul gol dar cu același nume va fi creat.

Uneori dorim să păstrăm fișierul vechi și să-i adăugăm date noi. Acestea se adaugă la sfârșitul celor existente. În Python folosim modul 'a' ca să deschidem un fișier căruia vrem să-i adăugăm date. Asta înseamnă că:

- dacă fișierul există deja, el nu va fi șters. Dacă nu există, va fi creat.
- când datele sunt scrise în fișier, ele vor fi adăugate la sfârșitul datelor existente.

De exemplu, să spunem că fișierul *barbati.txt* conține deja următoarele nume scrise fiecare pe o linie separată:

```
Costel  
Mirel  
Florel
```

Codul de mai jos deschide fișierul și adaugă următoarele date la conținutul existent:

```
myfile=open('barbati.txt', 'a')  
myfile.write('Gigel\n')  
myfile.write('Fanel\n')  
myfile.write('Stanel\n')  
myfile.close()
```

După aceasta vom avea numele anterior scrise (Costel, Mirel, Florel) la care se adaugă cele de mai sus. Fișierul va arăta în final așa:

```
Costel  
Mirel  
Florel  
Gigel  
Fanel  
Stanel
```

5.7. Scrierea și citirea datelor numerice

Șirurile pot fi scrise direct cu metoda *write*, în schimb numerele trebuie convertite în șiruri înainte de a fi scrise.

Python are o funcție preconstruită numită **str** care convertește o valoare într-un șir. Să spunem de pildă că variabilei *num* îi este atribuită valoarea 99. Expresia `str(num)` va returna șirul '99'.

Ex: *scrie_numere2.py*

```
#Programul demonstreaza cum numerele
#trebuie convertite in siruri inainte de a fi
#scrise in fisiere text
def main():
    #Deschide un fisier pentru scris
    outfile=open('numere.txt', 'w')
    #Ia trei numere de la user
    num1=int(input('Introdu un numar: '))
    num2=int(input('Introdu alt numar: '))
    num3=int(input('Mai baga unul: '))
    #Scrie cele trei numere in fisier
    outfile.write(str(num1) + '\n')
    outfile.write(str(num2) + '\n')
    outfile.write(str(num3) + '\n')
    #Inchide fisierul
    outfile.close()
    print('Date scrise in numere.txt')
#Cheama functia principala
main()
```

La ieșire, programul va afișa cele trei numere introduse de utilizator și mesajul *"Date scrise in numere.txt"*.

Expresia `str(num1) + '\n'` convertește valoarea referită de variabila *num1* într-un șir și o concatenează cu `'\n'` șirului. Când userul introduce să zicem valoare 50, expresia va produce șirul `'50\n'`. Drept rezultat, șirul `'50\n'` este scris în fișier.

5.8. Copierea unui fisier

Să presupunem că dorim să facem o copie a fișierului *barbati.txt* care se va chema *copybarbati.txt*. Pentru aceasta scriem urmatorul program (*copybarbati.py*):

```

1 fisier = open('barbati.txt', 'r')
2 linii = fisier.read()
3 fisier.close()
4 copie = open('copybarbati.txt', 'w')
5 copie.write(linii)
6 copie.close()
7 print('Copia fisierului a fost facuta')
8 copie = open('copybarbati.txt', 'r')
9 linii = copie.read()
10 print(linii)
11 copie.close()

```

Fig. 5.3. Copierea fisierului barbati.txt

Sa vedem si functionarea lui in Terminal:

```

192-168-0-100:desktop mirceaprodan$ python3 copybarbati.py
Copia fisierului a fost facuta
Costel
Mirel
Florel

192-168-0-100:desktop mirceaprodan$ █

```

Fig. 5.4. Functionare in Terminal

5.9. Fișiere binare

Următorul exemplu creează un fișier binar. Pentru aceasta folosim modul 'wb' (*binar.py*):

```

str = 'Salutare si voiosie!'
fisier = open('fisier_binar.bin', 'wb')
fisier.write(str.encode('utf-8'))
fisier.close()
fisier = open('fisier_binar.bin', 'rb')
fisiercontinut = fisier.read()
fisier.close()
print('Continutul fisierului este:')
print(fisiercontinut.decode('utf-8'))

```

Programul *binar.py* creează un fișier numit *fisier_binar.bin* în modul write și stochează în el șirul "Salutare si voiosie!". Șirul este codificat în sistemul UTF-8 înainte de fi scris în fișier. Fișierul este apoi închis. Ca sa confirmăm că șirul este stocat corect în fișier, îl deschidem cu modul read.

Să vedem cum funcționează:


```
192-168-0-100:desktop mirceaprodan$ python3 binar.py
Continutul fisierului este:
Salutare si voiosie!
192-168-0-100:desktop mirceaprodan$
```

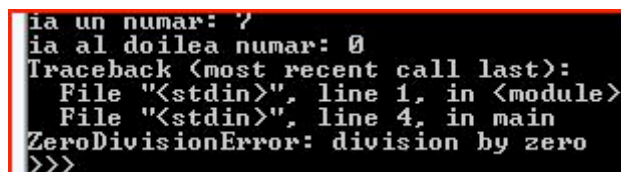
Fig. 5.5. Crearea unui fișier binar

5.10. Excepții

O excepție este o eroare care apare atunci când un program rulează și care are drept consecință oprirea lui brutală. Pentru mânuirea excepțiilor se folosește blocul de declarații **try/except**.

Programul de mai jos (*imparte.py*) oferă un exemplu.

```
#Programul imparte un numar la altul
def main():
    #Ia doua numere
    num1=int(input('ia un numar: '))
    num2=int(input('ia al doilea numar: '))
    #Imparte-le unul la celalalt si afiseaza rezultatul
    result=num1 / num2
    print(num1, 'impartit la', num2, 'este', result)
#Cheama main
main()
```



```
ia un numar: 7
ia al doilea numar: 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in main
ZeroDivisionError: division by zero
>>>
```

Fig.5.6. Eroare impartire la zero

El împarte două numere. Atunci când userul vrea să împartă un număr la zero, apare o excepție sau o eroare numită *traceback*. Ea dă informații cu privire la numărul liniei (liniilor) care cauzează excepția. Când se întâmplă să avem de a face cu o excepție, spunem că programul “a ridicat o excepție”. Ele pot fi prevenite scriind codul cu atenție.

Să rescriem codul de mai sus (*impartire2.py*) astfel încât excepția să nu mai apară:

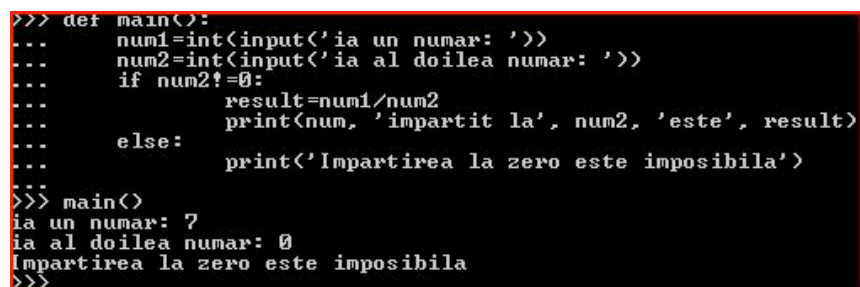
```
#Programul imparte un numar al altul
def main():
    #Ia doua numere
```

```

num1=int(input('ia un numar: '))
num2=int(input('ia al doilea numar: '))
#Daca num2 nu este 0, imparte num1 la num2
#si afiseaza rezultatul
if num2 != 0:
    result = num1 / num2
    print(num1, 'impartit la', num2, 'este', result)
else:
    print('Impartirea la zero este imposibila.')

#Cheama main
main()

```



```

>>> def main():
...     num1=int(input('ia un numar: '))
...     num2=int(input('ia al doilea numar: '))
...     if num2!=0:
...         result=num1/num2
...         print(num, 'impartit la', num2, 'este', result)
...     else:
...         print('Impartirea la zero este imposibila')
>>> main()
ia un numar: 7
ia al doilea numar: 0
Impartirea la zero este imposibila
>>>

```

Fig.5.7. Împărțirea la zero cu eroarea corectată

După cum poate v-ați dat seama, excepțiile trebuiesc pur și simplu ghicite atunci când programatorul scrie programul. El trebuie să-și puna mereu întrebarea *Ce s-ar întâmpla dacă?* (În cazul de mai sus, *Ce s-ar întâmpla dacă userul introduce valoarea celui de-al doilea număr ca fiind zero?*).

Tabelul de mai jos arata exceptiile care pot aparea atunci cand un program este evaluat de interpretor:

Excepție	Descriere
AssertionError	Apare atunci cand declaratia esueaza
AttributeError	Atributul nu este gasit in obiect
EOFError	Cand se incearca citirea dincolo de sfarsitul unui fisier
FloatingPointError	Apare cand operatia cu un numar in virgula mobila esueaza
IOError	Cand o operatie I/O esueaza

IndexError	Cand se foloseste un index aflat in afara gamei (range)
KeyError	Cand o cheie nu este gasita
OSError	Cand invocarea unui sistem de operare esueaza
OverflowError	Cand o valoarea este prea mare ca sa poata fi reprezentata
TypeError	Cand un argument de un tip nepotrivit este furnizat
ValueError	Cand valoarea unui argument nu este potrivita
ZeroDivisionError	Cand un numar se imparte la zero ori cand al doilea argument intr-o operatie modulo este zero

Tabel 5.1.

Capitolul VI Liste, tupluri, dicționare și seturi. Serializarea obiectelor (pickling)

6.1. Liste

O *listă* este un obiect care conține itemi multipli. Este similară *matricei* din alte limbaje de programare. Listele **sunt mutabile** ceea ce înseamnă că conținutul lor poate fi schimbat pe timpul execuției programului. Listele sunt structuri dinamice de date adică itemii lor pot fi adăugați sau șterși. Pentru aceasta se poate folosi *indexarea, felierea* (slicing) dar și alte metode de lucru cu liste.

Fiecare item dintr-o listă se cheamă **element** al listei.

Iată cum arată o declarație de creare a unei liste de întregi (ex):

```
numere_impere=[1, 3, 5, 7, 9]
```

Elementele listei sunt incluse între **paranteze drepte** și sunt despărțite între ele prin virgulă.

Să luăm un exemplu de listă cu șiruri:

```
nume= ['Mircea', 'Dana', 'Marcel', 'Stanel']
```

O listă poate să conțină tipuri diferite de valori:

```
lista = ['Stanel', 51, 4.157]
```

Lista "lista" conține un șir (Stanel), un întreg (51) și o valoare în virgulă mobilă (4.157).

Pentru afișarea conținutului listei folosim funcția `print ()` ca mai jos:

```
print (lista)
```

ceea ce va da la ieșire:

```
Last login: Mon Mar 18 21:46:53 on ttys000
You have mail.
192-168-0-100:~ mirceaprodan$ python
Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
>>> lista = ['Stanel', 51, 4.157]
>>> print(lista)
['Stanel', 51, 4.157]
>>> █
```

Fig.6.1. Listă de itemi cu valori diferite

Python are de asemenea o funcție preconstruită care convertește anumite tipuri de obiecte

în liste. Așa cum am vazut într-un capitol anterior, funcția `range` returnează un iterabil ce este un obiect care ține o serie de valori ce iterează peste ea.

Mai jos este un exemplu de folosire a funcției `range`:

```
numere = [5, 10, 15, 20]
numere = list(range(5))
```

Când executăm declarația se întâmplă următoarele:

- funcția `range` este invocată cu 5 drept argument; funcția returnează un iterabil care conține valorile 0, 1, 2, 3, 4.
- iterabilul este trecut ca argument funcției `list()`; funcția `list()` returnează lista [0, 1, 2, 3, 4].
- lista [0, 1, 2, 3, 4] este atribuită variabilei `numere`.

Operatorul de repetiție

Operatorul de repetiție face copii multiple unei liste și le pune laolaltă. Forma generală este:

```
lista * n
```

Un exemplu în care este multiplicată o listă de întregi dar și una cu valoare șir:

```
>>> lista = [1, 2, 3, 4]
>>> print(lista * 4)
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>>
>>>
>>> lista = ['Salut']
>>> print(lista * 5)
['Salut', 'Salut', 'Salut', 'Salut', 'Salut']
>>> █
```

Fig. 6.2. Multiplicarea listelor

Iterarea peste o listă cu bucla `for`

```
numere = [45, 67, 90, 3.45]
for n in numere:
    print(n)
```

Ceea ce la ieșire va da:

45

67

90

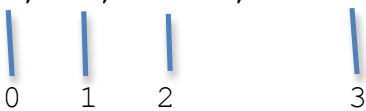
Indexarea

O metodă de accesare a elementelor individuale dintr-o listă este cu ajutorul unui *index*.

Fiecare element al listei are un index specific care-l poziționează în listă. Numărarea elementelor dintr-o listă începe de la zero (0), așa că primul element are indexul 0, al doilea are indexul 1, al treilea are indexul 2 s.a.m.d.

Să spunem că avem lista următoare:

```
lista = [12, 56, 6.345, 'Stanel']
```



Ca să accesăm elementul listei care are valoarea Stanel, utilizăm declarația:

```
print(lista[3]) .
```

Sau elementul cu indexul 1 al listei (care este 56).

```
>>> lista=[12,56,6.345,'Stanel']
>>> print(lista[3])
Stanel
>>> print(lista[1])
56
>>> █
```

Fig. 6.3. Acces index listă

Dacă folosim indecși negativi, vom identifica poziția elementelor relative la sfârșitul listei.

Așa se face că indexul -1 identifică ultimul element al listei, -2 pe penultimul și tot așa.

Indexul 0 este același cu indexul -4. Ce se întâmplă însă dacă vrem să aflăm elementul care numărat de la coada listei, are indexul -5 ?

Va apărea o eroare "Traceback" pentru că lista nu conține cinci indecși:

```
>>> print(lista[-1])
Stanel
>>> print(lista[-2])
6.345
>>> print(lista[-3])
56
>>> print(lista[0])
12
>>> print(lista[-4])
12
>>> print(lista[-3])
56
>>> print(lista[-5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> █
```

Fig.6.4. Afișarea elementelor cu ajutorul indecsilor negativi

Funcția len

Cu ajutorul funcției `len` putem afla lungimea unei liste.

Să luăm următorul cod:

```
lista = [1, 2, 3, 4, 5]
lungime = len(lista)
```

```
>>> lista=[1,2,3,4,5]
>>> len(lista)
5
>>> lista=[1,2,3,4,5]
>>> lungime=len(lista)
>>> print(lungime)
5
>>> █
```

Fig.6.5. Funcția `len()`

Funcția `len` poate fi folosită ca să prevină o excepție *IndexError* atunci când iterăm peste o listă cu o buclă:

```
lista = [1, 2, 3, 4, 5]
index = 0
while index < len(lista):
    print(lista[index])
    index +=1
```

Listele sunt mutabile

Așa cum scriam la începutul capitolului, listele în Python sunt mutabile ceea ce înseamnă că elementele ei pot fi schimbate.

Să vedem un exemplu:

```
lista = [1, 2, 3, 4, 5]          1
print(lista)                    2
lista[0] = 50                   3
print(lista)                    4
```

Cum funcționează:

Pe rândul 1 creăm lista "lista" iar apoi o afișăm pe rândul 2. Pe rândul 3, elementului 0 din listă – care este 1 – îi reatribuim valoarea 50. Cu linia 4 afișăm noua componență modificată

a listei:

```
>>> lista=[1,2,3,4,5]
>>> print(lista)
[1, 2, 3, 4, 5]
>>> lista[0] = 50
>>> print(lista)
[50, 2, 3, 4, 5]
>>> █
```

Fig.6.6. Listele pot fi modificate

Ce se întâmplă dacă nu folosim un index valid, ci unul care depășește numărul elementelor listei? Va apărea o excepție:

```
>>>
>>> #Acuma va apare o exceptie:
... lista=[1,2,3,4,5]
>>> lista[7] = 50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> █
```

Fig.6.7. Excepție de ieșire din marjă

Lista de mai sus are cinci elemente dar care au indecșii de la 0 la 4. Cum nu există index-ul 7, încercarea de modificare va eșua apărând o eroare Traceback.

Să luam un exemplu cu bucla while:

```
1 #Constanta NUMAR_ZILE tine numerele zilelor
2 #in care se aduna vanzarile unei saptamani
3 NUMAR_ZILE = 5
4
5 def main():
6     #creeaza o lista care tine vanzarile
7     #pentru fiecare zi
8     vanzari = [0] * NUMAR_ZILE
9
10    #creeaza o variabila care tine indexul
11    index = 0
12
13    print('Introdu vanzarile pentru fiecare zi.')
14
15    #Ia vanzarile din fiecare zi
16    while index < NUMAR_ZILE:
17        print('Zile #', index+1, ': ', sep='', end='')
18        vanzari[index] = float(input())
19        index += 1
20
21    #Afiseaza valoarea introdusa
22    print('Aici e valoarea introdusa: ')
23    for valoare in vanzari:
24        print(valoare)
25 #Cheama main
26 main()
```

Fig. 6.8. Program listă vânzări

Și acum ieșirea programului de mai sus:


```

192-168-0-100:desktop mirceaprodan$ python lista_vanzari.py
Introdu vanzarile pentru fiecare zi.
Zile #1:23.45
Zile #2:45.87
Zile #3:0.98
Zile #4:12345.5
Zile #5:45
Aici e valoarea introdusa:
23.45
45.87
0.98
12345.5
45.0
192-168-0-100:desktop mirceaprodan$ █

```

Fig.6.9. Ieșirea programului anterior

Concatenarea (legarea) listelor

Pentru concatenarea listelor în Python se folosește semnul + .

Cel mai bun mod de a înțelege cum funcționează este un exemplu:

```

lista1 = [8,9,10,11]
lista2 = [12,13,14,15]
lista3 = lista1 + lista 2

```

care va afișa la ieșire:

```

>>> lista1 = [8,9,10,11]
>>> lista2 = [12,13,14,15]
>>> lista3 = lista1 + lista2
>>> print(lista3)
[8, 9, 10, 11, 12, 13, 14, 15]
>>> █

```

Fig.6.10. Concatenarea listelor

La fel de bine putem folosi și concatenarea listelor cu valoare șir:

```

fete = ['Dana', 'Roxana', 'Ileana', 'Maria']
baieti = ['Costel', 'Gigel', 'Ion', 'Popica']
nume = fete + baieti

```

care va avea output-ul:

```

>>> fete = ['Dana','Roxana','Ileana','Maria']
>>> baieti = ['Costel','Gigel','Ion','Popica']
>>> nume=fete+baieti
>>> print(nume)
['Dana', 'Roxana', 'Ileana', 'Maria', 'Costel', 'Gigel', 'Ion', 'Popica']
>>> █

```

Fig.6.11. Concatenare șiruri

dar și tipuri diferite de valori (întregi, float, șir):

```
>>> fete = ['Marcela', 'Iona', 3.14]
>>> baieti=['Gigi', 57, 'Netoiu']
>>> nume=fete+baieti
>>> print(nume)
['Marcela', 'Iona', 3.14, 'Gigi', 57, 'Netoiu']
>>> █
```

Fig. 6.12. Tipuri diferite combinate într-o listă

Să reținem că putem combina *doar* **liste cu liste**. Dacă încercăm să combinăm o listă cu o altă entitate va apărea o excepție.

Felierea (slicing) listelor

Uneori ai nevoie să selectezi unele elemente dintr-o secvență dar nu pe toate. Atunci folosești felierea (*slice*). Ca să iei o porțiune dintr-o listă trebuie să scrii o declarație de forma:

```
lista[start : end]
```

unde *start* este indexul primului element din porțiune și *end* este indexul ultimului element din porțiune.

Să presupunem că avem următoarea listă:

```
zile_saptamana = ['Luni', 'Marti', 'Miercuri', 'Joi', 'Vineri',
                  'Sambata', 'Duminica']
```

Următoarea expresie folosește o porționare care preia elementele din lista de mai sus care au indexul 2 până la 5 (dar neincluzându-l pe cel din urmă):

```
zile_mijloc = zile_saptamana[2:5]
```

Când este executată declarația va rezulta:

```
['Marti', 'Miercuri', 'Joi']
```

sau un exemplu numeric:

```
>>> numere = [1,2,3,4,5,]
>>> print(numere)
[1, 2, 3, 4, 5]
>>> print(numere[1:3])
[2, 3]
>>> print(numere[3:4])
[4]
>>> print(numere[3:8])
[4, 5]
>>> █
```

Fig.6.13 Slicing

Dacă lăsăm gol locul unuia dintre indecși, Python va folosi automat 0 ca index de început.

```
>>> numere = [1,2,3,4,5,]
>>> print(numere)
[1, 2, 3, 4, 5]
>>> print(numere[:3])
[1, 2, 3]
>>> █
```

Fig. 6.14. Lipsa unui index

sau invers:

```
>>>
>>> print(numere[3:])
[4, 5]
>>> █
```

Fig. 6.15. Lipsa celuilalt index

Ce se întâmplă dacă lași goale spațiile indecșilor? Python va face o copie a întregii liste:

```
>>> numere = [1,2,3,4,5,]
>>> print(numere[:])
[1, 2, 3, 4, 5]
>>> █
```

Fig. 6.16. Lipsa ambilor indecși

Expresiile partajate pot avea și un pas (step value) ca în exemplul de mai jos:

```
>>> numere = [1,2,3,4,5,6,7,8,9,10]
>>> print(numere)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> print(numere[1:8:2])
[2, 4, 6, 8]
>>> █
```

Fig.6.17. Feliere cu pas

Pasul partajării este 2 ceea ce face ca porțiunea afișată să cuprindă fiecare al doilea număr din listă.

Găsirea itemilor dintr-o listă cu operatorul in

Forma generală este:

item in lista

Să luăm un exemplu (*in_lista.py*)

```
#demonstrarea operatorului in
def main():
    #creeaza o lista a numerelor produselor
    prod_num = ['V476', 'F890', 'Q143', 'R688']
    #Ia numarul unui produs pe care-l cauti
    cauta = input('Introdu numarul produsului: ')
    #afla daca numarul se gaseste in lista
    if cauta in prod_num:
        print(cauta, 'a fost gasit in lista.')
    else:
        print(cauta, 'nu a fost gasit in lista.')
#cheama functia principala
main()
```

6.2. Metode și funcții preconstruite pentru liste

Metoda append

Metoda `append` este folosită pentru adăugarea unui nou item listei. Itemul este trecut drept argument și este adăugat la sfârșitul listei.

Ex. program (*append.py*)

```
#acest program demonstreaza metoda append
#de adaugare a unui nou item intr-o lista
def main():
    #cream o lista goala
    nume_lista = [ ]
    #cream o variabila de control a buclei
    again = 'y'
    #adaugam cateva nume listei
    while again == 'y':
        #luam un nume de la utilizator
        nume = input('Introdu un nume: ')
        #adauga numele introdus in lista
        nume_list.append(nume)
        #mai adauga inca un nume
        print('Mai adaugi un nume? ')
        again = input('y = yes, anything else = no ')
    
```

```

        print()
    #afiseaza numele introduse
    print('Iata numele pe care le-ai introdus.')
    for nume in nume_lista:
        print(nume)
#cheama functia principala
main()

```

Metoda index

Uneori dorim să știm unde se află un item dintr-o listă. Pentru aceasta folosim metoda *index*. Trecem un argument metodei *index* și ea returnează index-ul primului element din listă care conține acel item. Dacă itemul nu este găsit, programul va ridica o excepție *ValueError*.

Ex. *mancare.py*

```

#programul demonstreaza cum luam un index al unui item
#dintr-o lista si apoi il inlocuim cu un alt item
def main():
    #cream o lista
    mancare = ['Pizza', 'Hamburg', 'Cipsuri']
    #afisam lista
    print(mancare)
    #luam itemul pe care il vrem inlocuit
    item = input('Ce item vrei sa schimbi? ')
    try:
        #ia indexul itemului din lista
        item_index = mancare.index(item)
        #ia valoarea de inlocuire
        noul_item = input('Introdu noua valoare: ')
        #inlocuieste vechiul item cu noul item
        mancare[item_index] = noul_item
        #afiseaza lista
        print('Aici este noua lista.')
        print(mancare)
    except ValueError:
        print('Itemul nu a fost gasit in lista.')

```

```
#cheama main
main()
```

Sa vedem cum functioneaza in Terminal:

```
192-168-0-100:desktop mirceaprodan$ python3 mancare.py
['Pizza', 'Hamburg', 'Cipsuri']
Ce item vrei sa schimbi? Pizza
Introdu noua valoare: Cartofi
Aici este noua lista.
['Cartofi', 'Hamburg', 'Cipsuri']
192-168-0-100:desktop mirceaprodan$
```

Fig. 6.18. Functionarea in Terminal

Metoda insert

Această metodă permite să adaugi un item într-o listă pe o poziție specifică. Ca să faci asta trebuie să treci două argumente metodei `insert`: un index care specifică locul unde itemul ar trebui să fie inserat și itemul pe care dorești să-l inserezi.

Ex.(*insert_list.py*)

```
def main():
    #creeaza o lista cu cativa itemi
    nume = ['Mircea', 'Dana', 'Auras']
    #afiseaza lista
    print('Lista inainte de inserare: ')
    print(nume)
    #insereaza un nume nou ca element 0 al listei
    nume.insert(0, 'Gigel')
    #afiseaza din nou lista
    print('Noua lista arata asa: ')
    print(nume)

#cheama functia main
main()
```

Metoda sort

Metoda `sort` rearanjează elementele unei liste așa încât ele să apară în ordine ascendentă.

Ex:

```
lista = [1,2 ,8,9,4,6,0]
print('Ordinea originala este:', lista)
lista.sort()
print('Lista sortata:', lista )
```

Metoda remove

```
def main():
    food = ['pizza', 'ciorba', 'cartofi']
    print(food)

    item=input('Ce item vrei sa sterg? ')

    try:
        food.remove(item)
        print('Iata lista refacuta: ')
        print(food)
    except ValueError:
        print('Itemul indicat n-a fost gasit!')

main()
```

Fig. 6.19. Metoda remove

Metoda reverse

După cum îi spune numele, metoda reverse inversează ordinea itemilor din listă.

Ex:

```
lista = [1,2,3,4,5]
lista.reverse()
print(lista)
```

```
>>> lista=[1,2,3,4,5]
>>> lista.reverse()
>>> print(lista)
[5, 4, 3, 2, 1]
>>> █
```

Fig. 6.20. Metoda reverse

Declaratia del

Cu această declarație ștergem un element din listă:

```
>>> lista=[1,2,3,4,5]
>>> del lista[3]
>>> print(lista)
[1, 2, 3, 5]
>>> █
```

Fig. 6.21. Declarația del

Funcțiile min și max

Ex:

```
>>>
>>> lista = [45,67,1,5,2,0,13]
>>> print('Valoarea minima este: ', min(lista))
Valoarea minima este:  0
>>> █
```

Fig. 6.22. Funcția min

La fel se întâmplă și cu max, înlocuind *min* cu *max*.

Totalul valorilor dintr-o listă

Pentru acest calcul trebuie să folosim o buclă *for* dar si un acumulator inițiat cu valoarea zero. Iată un exemplu:

```
#Acest program calculează totalul valorilor dintr-o listă
def main():
    #Cream lista
    numere = [2,3,6,8,10]
    #cream o variabila pe care o folosim drept acumulator
    total = 0
    #Calculam totalul elementelor listei
    for value in numere:
        total += value
    #Afisam totalul elementelor listei
    print('Totalul elementelor este', total)
#Invocam functia main
main()
```

Iar output-ul este: "Totalul elementelor este 30".

Media valorilor dintr-o listă

După ce calculăm totalul (ca mai sus) unei liste, ca să aflăm media valorilor din listă trebuie să împărțim totalul la lungimea listei. Ex. (*media.py*)

```
#Acest program calculează media valorilor dintr-o listă
def main():
    #creem lista
    scoruri = [2.5, 8.3, 6.5, 4.0, 5.2]
    #creem o variabila ca s-o folosim ca acumlator
    total = 0.0
```



```

#Calculam totalul valorilor din lista
for value in scoruri:
    total +=value

#Calculam media elementelor
media = total / len(scoruri)

#Afisam totalul elementelor listei
print('Media elementelor este', media)

#Invocam functia main
main()

```

Ieșirea programului este: “Media elementelor este 5.3”.

6.3. Tupluri

Un **tuplu** este o secvență imutabilă, ceea ce înseamnă că conținutul ei nu se poate schimba.

Un tuplu seamănă foarte mult cu o listă cu diferența că o dată creat, **elementele lui nu se pot schimba**. Elementele unui tuplu se închid **între o pereche de paranteze**, ca mai jos:

```

Python 3.3.1 (v3.3.1:d9893d13c628, Apr
tel)] on win32
Type "copyright", "credits" or "license
>>> tuplu = (1,2,3,4,5)
>>> print(tuplu)
(1, 2, 3, 4, 5)
>>>

```

Fig. 6.23. Elementele unui tuplu

Prima declarație creează un tuplu numit “tuplu” care conține elementele 1, 2, 3, 4, 5. A doua declarație afișează elementele tuplului.

Să vedem acum cum o buclă *for* iterează peste elementele unui tuplu:

```

>>>
>>> nume = ('Aurel', 'Daniela')
>>> for n in nume:
>>>     print(n)

Aurel
Daniela
>>>

```

Fig. 6.24. Buclă for în tupluri

De fapt, tuplurile suportă aceleași operații ca listele cu excepția celor care schimbă conținutul. Tuplurile suportă următoarele operații:

- Subscrierea indecsilor
- Metode ca `index`
- Funcții preconstruite: `len`, `min` și `max`
- Operații de feliere (slicing)
- Operatorul `in`
- Operatorii `+` și `*`

Tuplurile nu suportă metode ca `append`, `remove`, `insert`, `reverse` sau `sort`.

Când vrei să creezi un tuplu cu un singur element, acel element trebuie urmat neapărat de virgulă:

```
>>>
>>> tuplu = (5,)
>>> print(tuplu)
(5,)
>>>
```

Fig. 6.25. Tuplu cu un singur element

Motivul pentru care tuplurile există este acela că tuplurile sunt mai rapide ca listele.

Aceasta înseamnă că tuplurile sunt o bună alegere atunci când se procesează date foarte multe care nu urmează a fi modificate. Un alt motiv este acela că tuplurile sunt sigure.

Pentru că nu li se pot modifica elementele, nimeni nu poate să le manipuleze accidental.

Există două funcții care pot converti tuplurile în liste și invers. Acestea sunt: `list()` și `tuple()`:

```
>>>
>>> tuplu = (1,2,3)
>>> lista = list(tuplu)
>>> print(lista)
[1, 2, 3]
>>>
```

Fig. 6.26. Conversia tuplu-lista

6.4. Dicționare

Un dicționar e un obiect care stochează o colecție de date. Fiecare element dintr-un dicționar are două părți: o **cheie** și o **valoare**. Folosim cheia ca să localizăm valoarea.

Într-un dicționar clasic – ca de exemplu DEX, *cheia* este **cuvântul** pe care-l căutăm în dicționar ca să aflăm ce semnificație are. Spre exemplu, să luăm cuvântul “enoriaș” .

Conform DEX el înseamnă (are valoarea) “persoană credincioasă care ține de o parohie”.

Deci cheia este *enoriaș* iar valoarea este *persoană credincioasă care ține de o parohie*.

sau:

enoriaș = persoană credincioasă care ține de o parohie

Un alt exemplu este cartea de telefon în care cheia este numele persoanei pe al cărui număr de telefon dorim să-l aflăm iar numărul de telefon este valoarea.

Perechea *cheie-valoare* mai este referită și de expresia *cartografiere*.

Să creăm un dicționar:

```
carte_telefon = {'Mircea': '07225666', 'Gigel' : '076666111', 'Minel':  
'0744234567' }
```

Observăm că la crearea dicționarului `carte_telefon` folosim o pereche de acolade în interiorul căreia introducem *cheia* – care este numele persoanei – și numărul de telefon care este *valoarea*. Elementele dicționarului sunt fiecare cuprinse între ghilimele iar perechile sunt despărțite de virgule. Mai observăm că fiecare pereche își referă elementele prin semnul două puncte (:).

Primul element al dicționarului este `'Mircea' : '07225666'`;

Al doilea element este: `'Gigel' : '076666111'`;

Al treilea element este: `'Minel' : '0744234567'`.

În acest exemplu cheile și valorile sunt șiruri. Totuși, elementele unui dicționar pot fi de orice tip numai că – spre deosebire de liste, cheile sunt imutabile. Cheile pot fi șiruri, întregi, în virgulă mobilă sau tupluri. Cheile nu pot fi însa liste (am învățat deja că listele au elemente mutabile).

Extragerea unei valori din dicționar

Trebuie să reținem amănuntul că elementele dintr-un dicționar nu sunt stocate într-o ordine anume. Aceasta înseamnă că dicționarele nu sunt secvențe ordonate de date precum listele, tuplurile sau șirurile. Prin urmare nu putem folosi indexuri numerice ca să extragem valori. În loc de asta putem folosi cheile valorilor, nu însă înainte de a pune

înaintea lor numele dicționarului (în cazul de față “carte_telefon”):

```
>>> carte_telefon
{'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> carte_telefon['Florina']
'0250445566'
>>> █
```

Fig. 6.27. Extragerea unei valori dintr-un dicționar

În momentul în care solicităm o cheie care nu există în cartea de telefon, va apărea o eroare Traceback (KeyError):

```
>>> carte_telefon['Mircea']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Mircea'
>>> █
```

Fig. 6.28. KeyError

Folosirea operatorilor “in” și “not in” pentru testarea unei valori dintr-un dicționar

Ca să prevenim eroarea de mai sus putem folosi operatorul `in` dintr-o declarație `if` ca să vedem dacă o cheie există în dicționar:

```
>>> carte_telefon
{'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> carte_telefon = {'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> if 'Costel' in carte_telefon:
...     print(carte_telefon['Costel'])
...
0743123098
>>> █
```

Fig. 6.29. Prevenirea apariției KeyError

Declarația `if` determină dacă cheia “Costel” este în dicționarul `carte_telefon`. Dacă este, îi va afișa valoarea (numărul de telefon).

Putem de asemenea să folosim operatorul `not in` ca să determinăm dacă o cheie există într-un dicționar:

```
>>> carte_telefon = {'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> if 'Mircea' not in carte_telefon:
...     print('Mircea nu este in cartea de telefon')
...
Mircea nu este in cartea de telefon
>>> █
```

Fig. 6.30. Folosirea operatorului *not in*

Să reținem că șirurile care se compară cu ajutorul lui `in` și `not in` sunt case sensitive.

Adăugarea unor elemente dicționarului

Dicționarele sunt obiecte mutabile. Putem să le adăugăm noi perechi de elemente cu o declarație de următoarea formă:

```
nume_dictionar[cheie] = valoare

>>> carte_telefon = {'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> carte_telefon['Dorel'] = '0733567890'
>>> carte_telefon
{'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566', 'Dorel': '0733567890'}
>>>
```

Fig. 6.31. Adăugarea de elemente dicționarului

După cum se observă, în cartea noastră de telefon l-am adăugat pe “Dorel”.

Trebuie să mai reținem că într-un dicționar nu putem avea valori duplicate. Când atribuim o valoare unei chei existente, aceasta îi va lua locul celei vechi.

Ștergerea elementelor dintr-un dicționar

Forma generală este:

```
del nume_dictionar[cheie]
```

Ex:

```
>>> carte_telefon
{'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566', 'Dorel': '0733567890'}
>>> del carte_telefon['Dorel']
>>> carte_telefon
{'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> del carte_telefon['Cornelia']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Cornelia'
>>>
```

Fig. 6.32. Ștergerea elementelor dintr-un dicționar

Dacă privim atent exemplul de mai sus observăm că atunci când cheia pe care dorim să o excludem nu există în dicționar apare o eroare numită “KeyError”. În cazul de mai sus e vorba de cheia *Cornelia* care nu există în dicționar.

Aflarea numărului elementelor dintr-un dicționar

Pentru aceasta folosim funcția `len`:

```
>>> carte_telefon = {'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> numar_itemi = len(carte_telefon)
>>> print(numar_itemi)
3
>>> █
```

Fig. 6.33. Folosirea funcției len()

Amestecarea tipurilor de date într-un dicționar

Cheile unui dicționar sunt imutabile însa ele pot fi de orice tip: liste, întregi, numere în virgula mobilă, tuple.

Exemplele următoare ilustrează toate acestea:

```
>>> scor = {'Mircea': [88, 92, 99], 'Costel': [88, 92, 99], 'Ion': [90, 34, 55]}
>>> scor
{'Ion': [90, 34, 55], 'Costel': [88, 92, 99], 'Mircea': [88, 92, 99]}
>>> scor['Ion']
[90, 34, 55]
>>> costel_scor=scor['Costel']
>>> print(costel_scor)
[88, 92, 99]
>>> █
```

Fig. 6.34. Date de tipuri diferite într-un dicționar

sau:

```
>>> tipuri_amestecate={'abc':1, 987:'haleala', (3,4,5):[3,4,5]}
>>> tipuri_amestecate
{(3, 4, 5): [3, 4, 5], 'abc': 1, 987: 'haleala'}
>>> █
```

Fig. 6.35. Ibidem

Dicționar gol

Un dicționar gol se creează printr-o simplă declarație de forma generală:

```
>>> dictionar = { }
```

sau în cazul nostru

```
>>> carte_telefon = { }
```

După aceea putem să-i adăugăm elemente:

```
>>> carte_telefon['Ion'] = '0245345789'
>>> carte_telefon['Costel'] = '0743123098'
>>> carte_telefon['Florina'] = '0250445566'
```

Pentru crearea unui dicționar gol se mai poate folosi și funcția predefinită dict().

Ex:

```
carte_telefon = dict( )
```

Folosirea buclei for pentru iterarea peste elementele unui dicționar

```
>>> carte_telefon = {'Ion': '0245345789', 'Costel': '0743123098', 'Florina': '0250445566'}
>>> for cheie in carte_telefon:
...     print(cheie)
...
Ion
Costel
Florina
>>> for cheie in carte_telefon:
...     print(cheie, carte_telefon[cheie])
...
Ion 0245345789
Costel 0743123098
Florina 0250445566
>>> █
```

Fig.6.36. Folosirea buclei *for* pentru afișarea elementelor dicționarului

6.5. Metode ale dicționarelor

Obiectele unui dicționar au câteva metode cu care se pot manipula datele:

clear – curăță conținutul unui dicționar

get – ia valoarea asociată unei chei specifice. Dacă cheia nu este găsită, metoda nu ridică o excepție. În loc de asta, returnează valoarea implicită.

items - returnează toate cheile dintr-un dicționar și valorile lor asociate ca pe o secvență de tuple

keys – returnează toate cheile ca pe o secvență de tuple

pop – returnează valorile asociate cu o cheie specifică și le șterge (perechile cheie/valoare) din dicționar. Dacă cheia nu e găsită returnează valoarea implicită.

popitem – returnează o pereche întâmplătoare de cheie/valoare ca pe un tuple și o șterge din dicționar

values – returnează toate valorile din dicționar ca pe o secvență de tuple.

Să le luăm pe rând.

Metoda *clear*

Metoda *clear* șterge toate elementele unui dicționar, lăsându-l gol.

Formatul general al acestei metode este:

```
dictionary.clear( )
```

Ex:

```
>>> carte_telefon = {'Ion': '0245345789', 'Costel': '0743123098', 'Flori
0445566'}
>>> carte_telefon.clear()
>>> carte_telefon
{}
>>> █
```

Fig. 6.37. Metoda clear

Metoda get

Forma generală este:

```
dictionar.get(cheie, default)
```

în care *dictionar* e numele dicționarului, *cheie* este cheia pe care o căutăm în dicționar iar *default* este valoarea implicită pe care o returnează declarația în cazul în care cheia nu e găsită (în cazul de mai jos 'Valoarea n-a fost gasita').

Ex.:

```
>>> carte_telefon={'Ionel': '0722210848', 'Fana': '0760123456', 'Traian': '074455
6677'}
>>> valoare = carte_telefon.get('Dana', 'Valoarea n-a fost gasita')
>>> print(valoare)
Valoarea n-a fost gasita
>>> valoare = carte_telefon.get('Fana', 'Valoarea n-a fost gasita')
>>> print(valoare)
0760123456
>>> █
```

Fig. 6.38. Metoda get

Metoda items

Această metodă returnează toate cheile dicționarului și valorile asociate lor. Ea returnează valorile într-un tip special de secvență numit "vedere". Fiecare element din dicționar este un tuplu și fiecare tuplu conține o cheie cu valoarea ei asociată.

Ex:

```
>>> carte_telefon={'Ionel': '0722210848', 'Fana': '0760123456', 'Traian': '074455
6677'}
>>> carte_telefon.items()
dict_items([('Ionel', '0722210848'), ('Fana', '0760123456'), ('Traian', '0744556
677')])
>>> █
```

Fig. 6.39. Metoda items

Metoda keys

Această metodă returnează toate cheile dicționarului.

Ex:


```
>>> carte_telefon={'Ionel':'0722210848', 'Fana': '0760123456', 'Traian': '074455
6677'}
>>> carte_telefon.keys()
dict_keys(['Ionel', 'Fana', 'Traian'])
>>> █
```

Fig. 6.40. Metoda keys

Metoda pop

Metoda pop returnează valorile asociate cu o cheie specifică și șterge acele valori. În cazul în care nu găsește perechea solicitată cheie/valoare, afișează valoarea default.

Ex.:

```
>>> carte_telefon={'Ionel':'0722210848', 'Fana': '0760123456', 'Traian': '074455
6677'}
>>> carte_num=carte_telefon.pop('Mircea', 'Valoarea ceruta n-a fost gasita')
>>> carte_num
'Valoarea ceruta n-a fost gasita'
>>> carte_num=carte_telefon.pop('Traian', 'Valoarea ceruta n-a fost gasita')
>>> carte_num
'0744556677'
>>> carte_telefon
{'Ionel': '0722210848', 'Fana': '0760123456'}
>>> █
```

Fig. 6.41. Metoda pop

Metoda popitem

Metoda popitem returnează o pereche întâmplătoare de chei/valori și șterge acea pereche din dicționar.

Ex.:

```
>>> carte_telefon={'Ionel':'0722210848', 'Fana': '0760123456', 'Traian': '074455
6677'}
>>> carte_telefon
{'Ionel': '0722210848', 'Fana': '0760123456', 'Traian': '0744556677'}
>>> cheie, valoare = carte_telefon.popitem()
>>> print(cheie, valoare)
Ionel 0722210848
>>> carte_telefon
{'Fana': '0760123456', 'Traian': '0744556677'}
>>> █
```

Fig. 6.42. Metoda popitem

Metoda values

Această metodă returnează toate valorile dicționarului (fără cheiele lor) ca pe un dicționar.

Ex.:

```
>>> carte_telefon={'Ionel':'0722210848', 'Fana': '0760123456', 'Traian': '074455
6677'}
>>> carte_telefon.values()
dict_values(['0722210848', '0760123456', '0744556677'])
>>> █
```

Fig. 6.43. Metoda values

6.6. Seturi

Un set este un obiect care stochează o colecție de date. Un set are câteva caracteristici:

- Toate elementele setului sunt unice, adică doua elemente nu pot avea aceeași valoare
- Seturile sunt structuri neordonate, ceea ce înseamnă că elementele lui pot sta în orice ordine
- Elementele setului pot fi de diferite tipuri.

Pentru a crea un set, invocăm funcția preconstruită `set`:

```
setul_meu = set()
```

Aici aveam de-a face cu un set gol de elemente. Ca să-i adăugăm elemente, i le trecem drept argumente ale funcției `set()`:

```
setul_meu = set(['a', 'b', 'c'])
```

sau

```
setul_meu = set('abc')
```

```
>>> setul_meu=set(['a','b','c'])
>>> setul_meu
['a', 'b', 'c']
>>>
>>> setul_meu=set("abc")
>>> setul_meu
'abc'
>>>
```

Fig. 6.44. Crearea unui set in Python shell

Dacă scriem însă

```
setul_meu = set('a', 'b', 'c')
```

vom primi o eroare:

```
>>> setul_meu=set("a", "b", "c")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set expected at most 1 arguments, got 3
>>>
```

Fig. 6.45. Eroare

Aflarea numărului elementelor unui set se face prin intermediul funcției `len`:

```
setul_meu = set([1,2,3,4,5])
```

```
len(setul_meu)
```

```
>>> setul_meu=set([1,2,3,4,5])
>>> len(setul_meu)
5
>>>
>>>
```

Fig.6.46. Aflarea numărului elementelor unui set

Pentru adăugarea de elemente setului se utilizează metoda `add` (legată cu punct, desigur):

```
>>> setul_meu=set()
>>> setul_meu.add(1)
>>> setul_meu.add(2)
>>> setul_meu.add(3)
>>> setul_meu
{1, 2, 3}
>>>
```

Fig. 6.47. Metoda `add`

Se poate folosi și metoda `update`:

```
>>> setul_meu.update([4,5,6])
>>> setul_meu
{1, 2, 3, 4, 5, 6}
>>>
```

Fig. 6.48. Metoda `update`

Tipurile de date introduse pot fi diferite ca mai jos:

```
>>> setul_meu=set([1,2,3])
>>> setul_meu.update('abc')
>>> setul_meu
{1, 2, 3, 'c', 'b', 'a'}
>>>
```

Fig. 6.49. Date diferite

Pentru ștergerea elementelor unui set se pot folosi metodele `remove` sau `discard`. Itemul pe care-l vrem șters din set îl trecem drept argument al uneia dintre aceste metode. Spre exemplu, mai jos renunțăm la elementul 2 al setului:

```
>>> setul_meu
{1, 2, 3, 'c', 'b', 'a'}
>>> setul_meu.remove(2)
>>> setul_meu
{1, 3, 'c', 'b', 'a'}
>>>
```

Fig. 6.50. Metoda `remove` (am renunțat la numărul 2)

Sau `discard`:

```
>>> setul_meu.discard('a')
>>> setul_meu
{1, 3, 'c', 'b'}
```

Fig. 6.51. Metoda `discard` (am renunțat la litera a)

Ca să iterăm peste elementele unui set putem să folosim bucla `for`:

```

>>> setul_meu
{1, 3, 'c', 'b'}
>>>
>>> for val in setul_meu:
...     print(val)
...
1
3
c
b
>>>

```

Fig. 6.52. Bucla for utilizată în seturi

Uniunea a doua seturi conține elementele celor două seturi și se realizează cu metoda `union`:

```

>>>
>>> set1 = set([1,2,3,4,5])
>>> set2 = set([6,7,8,9])
>>> set3 = set1.union(set2)
>>> set3
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>

```

Fig. 6.53. Uniunea a două seturi

6.7. Serializarea obiectelor (pickling)

Serializarea unui obiect este procesul de convertire a obiectului într-un număr de biți care pot fi salvați într-un fișier ce poate fi accesat oricând după aceea. În Python, serializarea obiectelor se numește *pickling*³⁴.

Librăria standard a limbajului Python pune la dispoziție un modul numit **pickle** care este întrebuințat la serializarea obiectelor.

După ce este importat modulul *pickle*, facem următorii pași

- Deschidem un fișier pentru scrierea binară
- Invocăm metoda *dump* a modulul *pickle* cu care scriem în fișierul dat
- Închidem fisierul

Acuma, așa cum am văzut în capitolul anterior, ca să scriem în modul binar un fișier trebuie să folosim modul "wb" (*write binary*) atunci când invocăm funcția `open`:

```
outputfile = open('datele_mele.dat', 'wb')
```

Odată ce am deschis fișierul pentru scrierea binară, invocăm funcția *dump* a modulului *pickle*:

³⁴ to pickle = a mura, a băițui, a decapa, a afuma

```
pickle.dump(obiect, fișier)
```

unde `obiect` este o variabilă care referă obiectul pe care vrem să-l serializăm și `fișier` este o variabilă care face referință la obiectul fișier. După ce funcția este executată, obiectul referit de obiectul fișier este serializat și scris în fișier. Trebuie spus că se poate folosi modulul `pickle` pentru orice fel de obiecte: liste, tuple, dicționare, seturi, șiruri, întregi și numere în virgulă mobilă.

Într-un fișier putem salva oricâte obiecte serializate dorim. Când am terminat invocăm metoda `close` pentru închiderea fișierului.

Să luăm exemplul următor dintr-o sesiune interactivă:

```
>>> import pickle
>>> carte_telefon = {'Mircea': '0722423567'},
>>> carte_telefon = {'Mircea': '0722423567',
...                 'Dana': '0723665534',
...                 'Alex': '0767560912'}
>>> output_file = open('carte_telefon.dat', 'wb')
>>> pickle.dump(carte_telefon, output_file)
>>> output_file.close()
>>>
```

Fig. 6.54. Modulul Pickle si serializarea unui obiect

Ce se întâmplă mai sus?

Prima dată importăm modulul `pickle`. Apoi, creăm dicționarul `carte_telefon` cu numele drept chei și numerele de telefon ca valori. Mai departe, deschidem fișierul `carte_telefon.dat` pentru scriere binară. Următoarea linie de program invocă funcția `dump` a modulului `pickle` care serializează dicționarul `carte_telefon` și îl scrie în fișierul `carte_telefon.dat`. Ultima linie închide fișierul `carte_telefon.dat`.

Poate că la un moment dat avem nevoie să “desfacem” (*unpickle*) acel obiect. Pentru aceasta trebuie să facem următorii pași:

- Deschidem fișierul pentru citire binară
- Invocăm funcția `load` din modulul `pickle` ca să recuperăm un obiect din fișier
- După aceasta închidem fișierul

Ca să citim fișierul utilizăm modulul `'rb'`:

```
>>> import pickle
>>> input_file=open('carte_telefon.dat','rb')
>>> ct = pickle.load(input_file)
>>> ct
{'Dana': '0723665534', 'Alex': '0767560912', 'Mircea': '0722423567'}
>>> input_file.close()
>>>
```

Fig. 6.55. Citirea binară a fișierului

Ce se întâmplă? În prima linie importăm modulul `pickle`. În linia a doua deschidem fișierul `carte_telefon.dat` pentru citire binară. Apoi invocăm funcția `load` din modulul `pickle` ca să desfacem (`unpickle`) un obiect din fișierul `carte_telefon.dat`. Rezultatul îl atribuim variabilei `ct` (de la *carte de telefon*, voi puteți să-i spuneți cum vreți). Apoi afișăm cu comanda `ct` conținutul cărții de telefon. Ultima linie de cod închide fișierul `carte_telefon.dat`.

Capitolul VII Clase și obiecte. Programarea orientată pe obiect

După cum îi spune și numele, programarea orientată pe obiect este centrată pe *obiect*. Obiectele sunt create din date abstracte încapsulate și care funcționează împreună. Într-un program procedural, itemii de date sunt trecuți de la o procedură la alta. Separarea datelor și codului care operează datele poate rezolva problemele, dar astfel un program devine din ce în ce mai mare și mai complex.

Să presupunem că faci parte dintr-o echipă de programatori care scrie un program întins de baze de date. Programul este inițial proiectat astfel încât numele, adresa și numărul de telefon sunt referite de trei variabile. Sarcina ta este să proiectezi câteva funcții care acceptă cele trei variabile ca argument și să realizezi operații cu ele. Software-ul funcționează cu succes o perioadă, dar echipa ta este rugată la un moment dat să updateze soft-ul adăugându-i câteva noi componente. În timpul procesului de revizuire, programatorul șef îți spune că numele, adresa și numărul de telefon nu mai sunt stocate în variabilele știute. În loc de asta, ele urmează să fie stocate în liste. Asta înseamnă că trebuie să modifice toate funcțiile pe care le-ai scris, astfel încât să funcționeze cu liste în locul celor trei variabile. Făcând această modificare majoră, nu numai că vei avea enorm de muncă, ci deschizi oportunitatea ideală pentru apariția erorilor.

În timp ce programarea procedurală este centrată pe crearea de proceduri (funcții), programarea orientată pe obiect (OOP) este centrată pe crearea de obiecte.

7.1. Definiții

Un obiect este o entitate software care conține date și proceduri. Datele conținute într-un obiect sunt cunoscute drept *atributele* datelor obiectului. Atributele datelor obiect sunt variabile simple care referă date.

Procedurile prin care un obiect performează sunt cunoscute ca *metode*. Metoda unui obiect este funcția care realizează operații prin intermediul datelor obiectului.

Conceptual, obiectul este o unitate autoconținută care constă în atributele datelor și metodele ce le operează. OOP realizează separarea codului și datelor cu ajutorul

încapsulării și ascunderii datelor.

Încapsularea se referă la combinarea datelor și codului într-un singur obiect. Ascunderea datelor se referă la *abilitatea* unui obiect de a ascunde atributele datelor de codul aflat în afara obiectului. Doar metodele obiectului pot accesa direct și pot face schimbări asupra atributelor datelor obiectului.

De obicei, un obiect ascunde datele dar permite codului exterior să-i acceseze metodele. Cum vom vedea mai târziu, metodele obiectului oferă declarații din afara obiectului cu acces indirect asupra atributelor datelor.

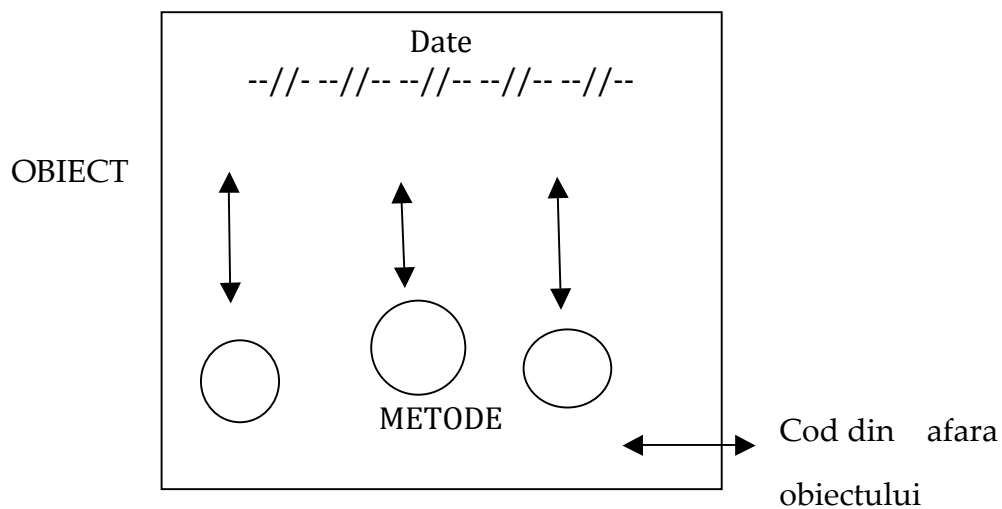


Fig. 7.1. Modelarea obiectelor

Când atributele datelor obiectului sunt ascunse codului extern și accesul atributelor datelor este restricționat de metodele obiectului, atributele sunt protejate de întreruperi sau defecțiuni accidentale. În plus, codul din afara obiectului nu trebuie să știe nimic despre structura și formatul intern al datelor obiectului. Codul are nevoie să interacționeze doar cu metodele obiectului. Acesta este motivul pentru care datele din obiect rămân neschimbate.

Un exemplu din viața de zi cu zi de obiect este ceasul cu alarmă pe care însă să încercăm să ni-l imaginăm ca pe un obiect software. El are următoarele atribute:

- secunda_curenta (o valoare cuprinsă între 0 și 59)
- minut_curent (o valoare cuprinsa intre 0 si 59)
- ora_curenta (o valoare cuprinsă între 0 și 12)
- timp_alarma (o oră și un minut valide)
- setare_alarma (True sau False)

După cum se poate vedea, atributele datelor sunt valori care definesc starea atunci când alarma ceasului e fixată. Tu, utilizatorul obiectului ceas cu alarmă nu poți manipula direct aceste date pentru că ele sunt private. Ca să schimbi valoarea datelor atribut trebuie să folosești una din metodele obiectului.

Iată mai jos câteva din metodele obiectului ceas cu alarmă:

- setare_timp
- setare_timp_alarma
- setare_alarma_on
- setare_alarma_off

Fiecare metodă manipulează unul sau mai multe atribute ale datelor. De exemplu, "setare_timp" îți permite să fixezi timpul alarmei ceasului. Activezi metoda prin apăsarea butonului din capul ceasului. Folosind alt buton (cheia) poți activa metoda "setare_timp_alarma". În plus, un alt buton îți permite să execuți metodele "setare_alarma_on" și "setare_alarma_off".

Trebuie să ții minte că aceste metode sunt activate de tine din afara ceasului cu alarmă.

Metodele care pot fi accesate de entități din afara obiectului sunt cunoscute ca **metode publice**.

Ceasul cu alarmă are de asemenea **metode private** care sunt părți private ale obiectului și care funcționează intern. Entitățile exterioare (ca tine, de exemplu), nu au acces direct la metodele interne ale ceasului cu alarmă. Obiectul este proiectat să execute aceste metode automat și să ascundă detaliile de tine.

Metodele private ale obiectului ceas cu alarmă sunt:

- incrementare_secunda_curenta
- incrementare_minut_curent

- incrementare_ora_curenta
- sunet_alarma

Metoda "incrementare_secunda_curenta" este executată în fiecare secundă. Aceasta schimbă valoarea atributului datei "secunda_curenta". Dacă atributul datei secunda_curenta este setat la 59 când metoda este executată, metoda este programată să reseteze valoarea "secunda_curenta" la 0 și face ca valoarea lui

"incrementare_minut_curent" să crească, adică această metodă este executată.

Metoda "incrementare_minunt_curent" adaugă 1 la atributul "minunt_curent", mai puțin când e setat la 59. În acest caz el resetează "minut_curent" la 0 și duce la execuția metodei "incrementare_ora_curenta". Metoda "incrementare_minut_curent" compară noul timp cu "setare_alarma". Dacă cele două coincid și alarma este pornită, metoda "sunet_alarma" este executată.

7.2. Clase

O *clasă* este un cod care specifică atributele datelor și metodele pentru un tip particular de obiect.

Înainte ca un obiect să fie creat, el trebuie să fie proiectat de un programator.

Programatorul determină atributele și metodele necesare și apoi crează o clasă. Să ne gândim la o clasă ca la un șablon după care obiectele sunt create. Șablonul servește același scop ca acela folosit la proiectarea unei case. Șablonul în sine nu este o casă dar el descrie în detaliu o casă. Când folosim un șablon ca să construim o casă putem spune că *construim o instanță* a casei descrise de șablon. Dacă vrem, putem să construim oricâte case (identice) dorim după șablon. Fiecare casă este o instanță separată a casei descrise de șablon.

Programatorii au mai imaginat și o altă asemănare pentru a descrie mai bine diferențele dintre clase și obiect. Spre exemplu o formă de prăjitură și prăjitura în sine. Sau, dacă mă gândesc la gogoșile pe care le făcea bunica mea cu paharul, atunci gura paharului descria o gogoasă dar nu era o gogoasă la rândul ei. Forma de prăjitură este deci folosită să faci prăjituri. Să ne gândim la o clasă ca la un șablon, o formă de prăjituri sau gura unui pahar iar la obiectele create cu ele ca la prăjituri, gogoși sau case.

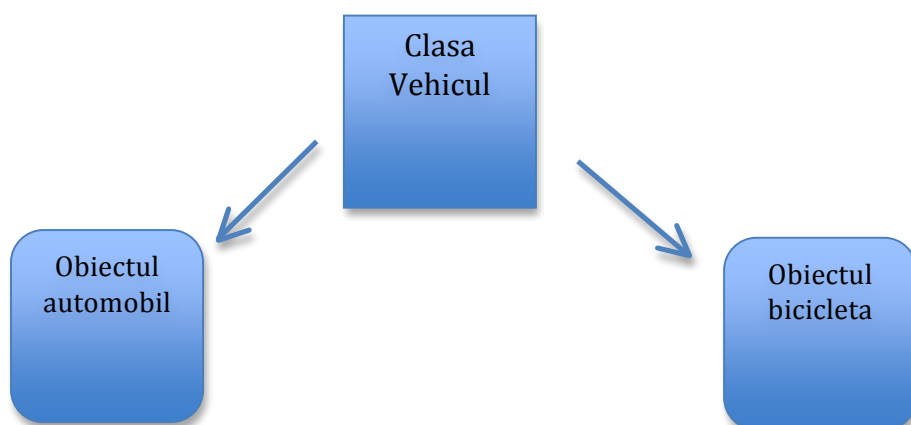
Un alt exemplu foarte nimerit de clasă este formularul de la ghișeul primăriei prin care soliciți ceva. Formularul este proiectat o singură dată iar apoi este multiplicat în mii de exemplare. Solicitantul (cetățeanul) vine la primărie cu o problemă și i se înmânează sau își ia singur un formular pe care îl completează. Fiecare cetățean are datele lui personale pe care le introduce. O dată formularul completat avem deja de a face cu o **instanță** a clasei *formular*.

Deci o clasă descrie caracteristicile unui obiect. Când un program rulează el poate folosi o clasă ca să creeze în memorie oricâte obiecte dorește. *Fiecare obiect care este creat de o clasă se cheamă instanța clasei.*

De exemplu, Andrei se ocupă cu scrierea de programe de calculator. El proiectează un program care să catalogheze diversele tipuri de automobile. Ca parte a programului, Andrei creează o clasă numită Vehicul care are caracteristicile comune oricăror vehicule (tip, producător, preț etc).

Clasa Vehicul specifică obiectele ce pot fi create din ea. Apoi, Andrei scrie declarațiile de program care creează un obiect numit *automobil* care este o instanță a clasei Vehicul.

Obiectul automobil este o entitate care ocupă un loc în memoria computerului și stochează date despre automobil. El are atributele și metodele specific clasei Vehicul. Apoi Andrei scrie o declarație care creează un obiect numit *bicicleta*. Obiectul bicicleta este de asemenea o instanță a clasei Vehicul. El are propriul loc în memorie și stochează date despre bicicleta. Altfel, obiectele automobil și bicicleta sunt două entități separate în memoria computerului, ambele create din clasa Vehicul. Asta înseamnă că fiecare dintre cele două obiecte are atributele și metodele descrise de clasa Vehicul.



7.3. Crearea claselor în Python

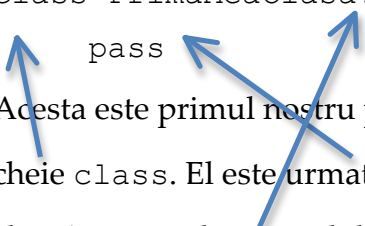
Nu trebuie să scriem prea mult cod ca să ne dăm seama că Python este un limbaj foarte curat. Când vrem să facem ceva, *facem* pur și simplu, fără a urma prea mulți pași.

Omniprezentul “hello world” în Python, este după cum știm, afișat cu o singură linie de cod.

În mod similar, cea mai simplă clasă în Python 3 arată cam așa:

```
class PrimaMeaClasa:
```

```
    pass
```



Acesta este primul nostru program obiect orientat! Definiția clasei începe cu cuvântul cheie `class`. El este urmat de un nume (la alegerea noastră) care identifică clasa (numele clasei) și apoi de semnul două puncte. Numele clasei trebuie să urmeze regulile standard de denumire a variabilelor. De asemenea este recomandat ca numele să fie scris în stilul cocoasă de cămilă (CamelCase): să înceapă cu o literă mare iar apoi fiecare cuvânt subsecvent la fel (ex: **PrimaMeaClasa**).

Linia de definiție a clasei este urmată de conținutul indentat al clasei. Ca și alți constructori din Python, indentarea este folosită ca să delimiteze clasele în locul acoladelor din alte limbaje de programare bazate pe “C”.

Pentru că prima noastră clasă nu face practic nimic, folosim cuvântul cheie `pass` ca să arătăm că nu urmează nicio acțiune (ca la poker). Ne putem gândi că nu sunt prea multe de făcut cu o clasă de bază, dar ea permite să instanțiezi obiecte după acea clasă. Putem să încărcăm clasa în interpretorul Python 3 ca să ne jucăm cu ea interactiv. Ca să facem asta, salvăm definiția clasei de mai devreme într-un fișier numit `first_class.py` și apoi rulăm comanda:

```
python -i first_class.py.35
```

Argumentul `-i` îi spune lui Python să ruleze codul și apoi “să-l arunce” interpretorului interactiv. Următoarea sesiune în interpretor demonstrează interacțiunea de bază cu o clasă:

```
>>> a = PrimaMeaClasa()
```

³⁵ Cu această ocazie învățăm și cum să trecem din modul script în modul interactiv din linia de comandă

```

>>> b = PrimaMeaClasa()
>>> print(a)
< __main__.MyFirstClass object at 0xb7b7faec>
>>> print(b)
< __main__.MyFirstClass object at 0xb7b7fbac>
>>>

```

Explicații

Codul de mai sus instanțiază două obiecte ale clasei *MyFirstClass*, obiecte numite *a* și *b*.

Crearea instanței unei clase înseamnă de fapt scrierea numelui clasei urmată de o pereche de paranteze. Ea arată aproape la fel ca invocarea unei funcții normale, numai că Python *știe* că invocă o clasă și nu o funcție, deci înțelege că sarcina lui e să creeze un obiect nou. Când sunt afișate, cele două obiecte ne spun despre ce clasă e vorba și care e adresa de memorie unde ele sălășluiesc. Adresele de memorie nu sunt prea mult folosite în Python, dar în exemplul anterior ele demonstrează că sunt implicate două obiecte distincte.

7.4. Adăugarea atributelor

În programarea OOP există două denumiri frecvent întâlnite: **metode** și **atribute**.

Atributele sunt de fapt *variabilele* din programarea procedurală iar metodele sunt *funcțiile*.

În exemplul nostru avem deci o clasă de bază dar total inutilă. Ea nu conține nicio dată și mai ales, nu face nimic. Ce se întâmplă dacă oferim un atribut unui obiect dat?

Putem să setăm un atribut arbitrar unui obiect instanțiat folosind *notația cu punct* (*dot notation*):

```

class Punct:
    pass
p1 = Punct()
p2 = Punct()

p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)

```

Dacă rulăm acest cod, cele două declarații de afișare de la sfârșit, ne dau noile valori ale atributelor celor doua obiecte:

```
5 4
```

```
3 6
```

Ce face totuși acest cod? El creează o clasă goală numită `Punct` care nu are date și nici comportamente. Apoi, el creează două instanțe ale clasei și atribuie fiecăreia coordonatele `x` și `y` care identifică un punct bidimensional. Tot ceea ce avem de făcut este să atribuim o valoare atributului obiectului folosind sintaxa generală:

```
<obiect>.<atribut> = <valoare>
```

Aceasta este uneori denumită *notația cu punct (dot notation)*. Valoarea poate fi orice: o primitivă Python, a data preconstruită etc. Poate fi chiar o funcție sau un alt obiect.

7.5. Să punem clasa la treabă

Programarea Orientata pe Obiect (OOP) este despre interacțiunea obiectelor între ele.

Suntem interesați așadar să invocăm acțiuni care fac ca lucrul acesta să se întâmple. Este timpul deci să adăugăm comportamente claselor noastre.

Să concepem modelul unor acțiuni ale clasei `Punct`. Putem începe cu o metodă numită `reset` care mută punctul la origine (originea este punctul unde valoarea lui `x` și `y` este egală cu zero, v-amintiți probabil de la algebră de *ordonata* și *abscisa*). Aceasta este o bună introducere pentru că ea necesită orice parametru:

(*punct.py*)

```
class Punct:
    def reset(self):
        self.x = 0
        self.y = 0
p = Punct()
p.reset()
print(p.x, p.y)
```

Declarația `print` afișează cele două zerouri ale atributelor `(0 0)` ca în sesiunea interactivă de mai jos:

```

>>> class Punct:
...     def reset(self):
...         self.x = 0
...         self.y = 0
...
>>> p = Punct()
>>> p.reset()
>>> print(p.x, p.y)
0 0
>>>

```

Fig.7.2. Ieșire program *punct.py*

O metodă în Python este identică cu definirea unei funcții. Ea începe cu cuvântul cheie `def` urmat de un spațiu și de numele metodei. Aceasta e urmată de o pereche de paranteze care conțin parametrul *self* (vorbit imediat despre el) și se termină cu două puncte.

Următoarea linie este indentată și conține declarațiile din interiorul metodei. Aceste declarații pot fi cod Python arbitrar care operează asupra obiectelor însăși precum și orice parametru trecut metodei.

7.6. Argumentul *self*

Singura diferență dintre metode și funcțiile normale este aceea că fiecare metodă necesită un argument. Acest argument este convențional numit ***self***.

Argumentul `self` dintr-o metodă este o simplă referință la un obiect a cărui metodă a fost invocată. Prin el putem accesa atribute și metode ale acelui obiect. Aceasta este exact ceea ce facem înăuntrul metodei `reset` atunci când setăm atributele `x` și `y` obiectului `self`.

Să notăm că atunci când invocăm metoda `p.reset()`, nu trebuie să trecem argumentul *self* în ea. Python are automat grijă să facă acest lucru. El știe când invocăm o metodă a obiectului `p`, deci el trece automat acel obiect metodei.

Oricum, o metodă chiar este cu adevărat doar o funcție care se întâmplă să fie într-o clasă.

În loc să invocăm metoda pe un obiect, putem invoca funcția clasei, trecând explicit obiectul ca argumentul `self`:

```

p = Punct()
Punct.reset(p)
print(p.x, p.y)

```

Ieșirea este aceeași ca la exemplul anterior, pentru că se desfășoară același proces.

Sa adăugăm o nouă metodă care permite să muți un punct într-o poziție arbitrară , nu doar la origine. Putem să includem un alt obiect Punct ca input și apoi să returnăm distanța dintre ele:

```
import math
class Punct:
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)
    def calculeaza_distanța(self, alt_punct):
        return math.sqrt(
            (self.x - alt_punct.x)**2 +
            (self.y - alt_punct.y)**2)
#cum folosim programul:
punct1 = Punct()
punct2 = Punct()

punct1.reset()
punct2.move(5, 0)
print(punct2.calculeaza_distanța(punct1))
assert (punct2.calculeaza_distanța(punct1) ==
        punct1.calculeaza_distanța(punct2))
punct1.move(3, 4)
print(punct1.calculeaza_distanța(punct2))
print(punct1.calculeaza_distanța(punct1))
```

7.7. Definirea unei clase – o altă abordare

Ca să creezi o clasă trebuie să scrii definiția ei. Definiția clasei este un set de declarații care stipulează metodele și atributele datelor obiectului.

Să luăm un exemplu simplu. Să presupunem că scriem un program care simulează întoarcerea unei monede. Avem nevoie să repetăm întoarcerea monedei ca să determinăm de fiecare dată dacă e “cap” sau “pajură”.

Folosind OOP vom scrie o clasă numită Moneda care descrie comportamentul aruncării unei monede.

Iată mai jos programul *moneda.py*:

```
import random 1
#Clasa Moneda simuleaza o moneda
#care poate fi intoarsa
class Moneda: 4
    #Metoda __init__ initializeaza attributele
    # partii de sus a monedei (in_sus) cu 'Cap' 6
    def __init__(self):
        self.in_sus = 'Cap' 8
    #Metoda intoarce genereaza un numar aleator
    # in gama 0 si 1. Daca numarul este 0 atunci
    # in_sus (partea de sus) este setata pe 'Cap'
    # altfel, in_sus este setata pe 'Pajura' 12
    def intoarce(self): 13
        if random.randint(0, 1) == 0: 14
            self.in_sus = 'Cap' 15
        else: 16
            self.in_sus = 'Pajura' 17
    #Metoda ia_in_sus returneaza o valoare
    #referita de in_sus 19
    def ia_in_sus(self): 20
        return self.in_sus
def main():
    #cream un obiect (o instantă) din clasa Moneda
    moneda_mea = Moneda()
    #afisam partea monedei care este in sus
    print('Aceasta parte este in sus: ', moneda_mea.ia_in_sus())
    #intoarcem moneda
    print('Intorc moneda')
    moneda_mea.intoarce()
    #afisam fata monedei aflata in sus
    print('Aceasta e in sus:', moneda_mea.ia_in_sus())
main()
```

Ce se întâmplă? În linia 1 importăm modulul *random*, căci nu-i așa, “dăm cu banul”, adică urmează o valoare întâmplătoare. El este necesar pentru că folosim funcția *randint* inclusă în modulul *random* ca să generăm un număr aleator.

Linia 4 reprezintă începutul definiției clasei. Ea începe cu cuvântul cheie `class` urmat de numele clasei (care este `Moneda`), urmat la rândul lui de două puncte (`:`). Aceleași reguli care se aplică la numele unei variabile, sunt valabile și aici. Să notăm totuși că numele clasei începe cu majusculă: `Moneda`. Nu este obligatoriu, dar este o convenție general utilizată de programatori. Ea ajută să facem distincția între numele claselor și numele variabilelor atunci când citim codul.

Clasa `Moneda` conține trei metode:

metoda `__init__` care apare în liniile 6 și 7

metoda `intoarce` care ocupă liniile 12-16

metoda `ia_in_sus` din liniile 19 și 20

Să privim mai atent începutul fiecărei metode și să observăm că fiecare dintre ele are un parametru numit `self`:

Linia 6 `def __init__ (self) :`

Linia 12 `def __intoarce__ (self) :`

Linia 20 `def ia_in_sus (self) :`

Parametrul `self` este obligatoriu în fiecare metodă a unei clase. Când o metodă este executată ea trebuie să aibă o cale ca să știe asupra cărui atribut al datelor obiectului trebuie să opereze. Aici este momentul în care parametrul `self` intră în rol. Când o metodă este invocată, Python face ca parametrul `self` să refere un obiect specific asupra căruia se presupune că metoda operează.

Să privim mai atent fiecare dintre metode. Prima metodă numită `__init__` este definită în liniile 6 și 7:

```
def __init__ (self):  
    self.in_sus = 'Cap'
```

Majoritatea claselor în Python au o metodă specială numită `__init__` care este executată automat atunci când o instanță a clasei este creată în memorie. Metoda `__init__` este mai cunoscută sub numele de *metoda de inițializare* pentru că inițializează atributele datelor obiectului. Numele metodei începe cu două liniuțe joase urmate de cuvântul `init` și apoi de încă două liniuțe joase.

Imediat după ce un obiect este creat în memorie, metoda `__init__` este executată și parametrul `self` este automat atribuit obiectului ce tocmai a fost creat.

Avem apoi declarația din linia 14 care este executată:

```
self.in_sus = 'Cap'
```

Această declarație atribuie șirul `'Cap'` atributului `in_sus` care aparține obiectului ce tocmai a fost creat. Drept rezultat al metodei `__init__` fiecare obiect pe care-l creăm din clasa `Moneda` va avea un atribut `in_sus` care e setat pe `'Cap'`.

Nota: Metoda `__init__` este de obicei **prima metoda** înăuntrul definiției clasei.

Metoda `intoarce` apare în liniile 12 – 16:

```
def intoarce(self):  
    if random.randint(0, 1) == 0:  
        self.in_sus = 'Cap'  
    else:  
        self.in_sus = 'Pajura'
```

Și această metodă necesită variabila parametru `self`. Când metoda `intoarce` este invocată, `self` referă automat obiectul asupra căruia metoda operează. Metoda `intoarce` simulează întoarcerea unei monede. Când metoda este invocată, declarația `if` cheamă funcția `random.randint` ca să dea un întreg situat între 0 și 1. Dacă numărul este 0 atunci declarația următoare atribuie `'Cap'` lui `self.in_sus`. Altfel, declarația atribuie `'Pajura'` lui `self.in_sus`.

Metoda `ia_in_sus` apare în liniile 19 și 20:

```
def ia_in_sus(self):  
    return self.in_sus
```

Încă o dată, metoda necesită parametrul `self`. Această metodă returnează valoarea lui `self.in_sus`. Chemăm această metodă ori de câte ori vrem să știm care față a monedei este în sus.

În imaginea următoare avem ieșirea programului `moneda.py` rulat în Command Prompt:

```

C:\Users\duke2007\Desktop>python moneda.py
Aceasta este in sus: Cap
Intorc moneda
Aceasta este in sus: Pajura
C:\Users\duke2007\Desktop>

```

Fig. 7.3. Ieșire program *moneda.py*

Să vedem în continuare și cum funcționează o clasă. Astfel, vom scrie clasa *Asigurari* care ține câteva informații despre firme de asigurări (*asigurari.py*):

```

#Clasa Asigurari tine informatii despre
#firme de asigurari
class Asigurari:

    #Metoda __init__ initializeaza attributele
    def __init__(self, firma, pret):
        self.__firma = firma
        self.__pret = pret

    #Metoda set_firma accepta un argument
    #pentru firma de asigurari
    def set_firma(self, firma):
        self.__firma = firma

    #Metoda set_pret accepta un argument
    #pentru pretul asigurarii
    def set_pret(self, pret):
        self.__pret = pret

    #Metoda ia_firma returneaza numele firmei
    #de asigurari
    def ia_firma(self):
        return self.__firma

    #Metoda ia_pret returneaza
    #pretul politei de asigurare
    def ia_pret(self):
        return self.__pret

```

Si acum programul care o testeaza (*polita.py*):

```

import asigurari

def main():
    firma = input('Introdu firma de asigurari')
    pret = float(input('Introdu pretul asigurarii'))

    #Creeaza o instanta a clasei Asigurari
    asigurare = asigurari.Asigurari(firma, pret)

    #Afiseaza datele introduse

```

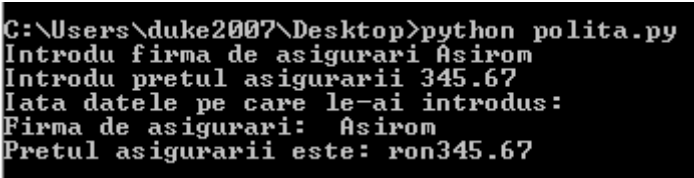
```

print('Iata datele pe care le-ai introdus: ')
print('Firma de asigurari: ', asigurare.ia_firma())
print('Pretul asigurarii este: ron',\
      format(asigurare.ia_pret(), ',.2f'), sep=' ' )

main()

```

În final, să testăm funcționarea în linia de comandă a programului *polita.py*:



```

C:\Users\duke2007\Desktop>python programa.py
Introdu firma de asigurari Asiom
Introdu pretul asigurarii 345.67
Iata datele pe care le-ai introdus:
Firma de asigurari: Asiom
Pretul asigurarii este: ron345.67

```

Fig. 7.4. Ieșirea programului *polita.py*

Foarte important! Clasa Asigurari (*asigurari.py*) se importă fără terminația (extensia) *.py*. Clasa (*asigurari.py*) se salvează întotdeauna **în același director/folder** (în cazul meu pe Desktop) în care salvăm și programul care o folosește (*polita.py*). Altfel, programul nu va funcționa.

7.8. Moștenirea (Inheritance) în Python

Moștenirea reprezintă o tehnică de copiere a datelor și funcțiilor unei clase existente într-o altă clasă. În loc să pornim de la zero, o clasă existentă poate fi moștenită de o alta căreia îi adăugăm date noi. Clasa moștenită se cheamă *clasă de bază* sau *superclasă* iar clasa moștenitoare se cheamă clasă *derivată* sau *subclasă*. Trebuie menționat că clasa care moștenește preia toate proprietățile clasei de bază.

Există trei tipuri de moștenire. Acestea sunt:

1. moștenirea simplă
2. moștenirea pe mai multe niveluri
3. moștenirea multiplă

Noi vom vorbi în continuare despre moștenirea simplă.

Toate clasele din limbajul Python sunt derivate dintr-o clasă specială (părinte) numită *Object*. Chiar dacă nu este explicitat scrisă, ea oricum este implicită. Să zicem de exemplu că vrem să creăm o clasă *Animal* pe care o derivăm din clasa *Object* (*animal.py*):

```
class Animal(Object):
```

Să luăm în continuare un exemplu.

#Aici avem superclasa

```
class Animal(object36):  
    vorbesteStr = 'Salut de la Animal!'  
    pass
```

#Aici avem clasa derivata

```
class Caine(Animal):  
    pass  
  
latra = Caine.vorbesteStr  
print(latra)
```

La ieşire vom avea „Salut de la Animal!”.



```
type "help", "copyright", "credits" or "license()">>> class Animal(object):  
...     vorbesteStr = 'Salut de la Animal!'  
...     pass  
...  
>>> class Caine(Animal):  
...     pass  
...  
>>> latra = Caine.vorbesteStr  
>>> print(latra)  
Salut de la Animal!  
>>>
```

Fig. 7.5. Ieşire program *animal.py*

Clasa *Caine* moşteneşte attributele clasei *Animal* şi afişează şirul pe care clasa *Animal* l-a avut ca atribut. Dacă însă vrem să suprascriem valoarea unui atribut din clasa derivată

Caine, facem astfel (*caine.py*):

```
class Animal(object):  
    vorbesteStr = 'Salut de la Animal!'  
    pass  
  
class Caine(Animal):  
    vorbesteStr = 'Salut de la caine!'  
    pass  
  
latra = Caine.vorbesteStr  
print(latra)
```

care va avea ieşirea „Salut de la caine” aşa cum se poate vedea în figura de mai jos:

³⁶ Putem să adăugăm sau nu *object*, functionarea programului nu va fi afectată.

```

>>> class Animal(object):
...     vorbesteStr = 'Salut de la Animal!'
...     pass
>>> class Caine(Animal):
...     vorbesteStr = 'Salut de la caine!'
...     pass
>>> latra = Caine.vorbesteStr
>>> print(latra)
Salut de la caine!
>>>

```

Fig. 7.6. Ieșirea programului caine.py

Putem de asemenea să executăm o metodă din clasa părinte (superclasă) înăuntrul unei subclase așa cum se poate vedea în următorul program rulat în interpretorul Python:

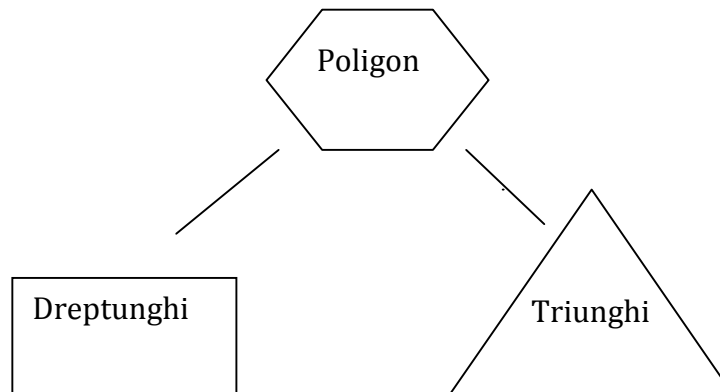
```

>>> class Animal(object):
...     def habitat(self):
...         vorbesteStr = 'Salut de la animal!'
...         print(vorbesteStr)
>>> class Caine(Animal):
...     pass
>>> latra = Caine()
>>> print(latra.habitat())
Salut de la animal!
None
>>>

```

Fig. 7.7. Metoda înăuntrul subclasei

Să mai luăm un exemplu - de această dată din geometrie. Să spunem că dorim să aflăm suprafața unui dreptunghi și a unui triunghi care sunt la origine poligoane.



Prin urmare trebuie să creăm superclasa Poligon și apoi subclasele Dreptunghi și Triunghi care să o moștenească.

Prima dată creăm superclasa Poligon (*poligon.py*):

```

class Poligon:
    lungime = 0
    latime = 0
    def set_valori(self, lungime, latime):

```

```
Poligon.lungime = lungime
Poligon.latime = latime
```

Apoi subclasa Dreptunghi care moștenește caracteristicile superclasei Poligon (*Dreptunghi.py*):

```
from poligon import*
class Dreptunghi(Poligon):
    def suprafata(self):
        return self.lungime * self.latime
```

În continuare creăm clasa Triunghi (*Triunghi.py*):

```
from poligon import*
class Triunghi(Poligon):
    def suprafata(self):
        return(self.lungime * self.latime) / 2
```

În final, scriem programul necesar calculării celor două suprafețe (*mostenire.py*):

```
from Dreptunghi import*
from Triunghi import*

#cream o instanta pentru fiecare subclasa
drept = Dreptunghi()
tri = Triunghi()
#invocam metoda mostenita de la superclasa, trecandu-i
#argumentele care sunt atribuite variabilelor clasei
drept.set_valori(35, 25)
tri.set_valori(16, 20)
#afisam rezultatele manipuland variabilele
#clasei mostenite din clasa de baza
print('Aria dreptunghiului este: ', drept.suprafata())
print('Aria triunghiului este: ', tri.suprafata())
```

La ieșire vom avea:


```
C:\Users\duke2007\Desktop\fisiere_carte>python mostenire.py
Aria dreptunghiului este: 320
Aria triunghiului este: 160.0
C:\Users\duke2007\Desktop\fisiere_carte>
```

Fig. 7.8. Output program *mostenire.py*

Nu știu dacă acest capitol a fost indeajuns de explicit, prin urmare m-am gândit să dau un exemplu de aplicație foarte interesantă scrisă în Python, care are o excelentă parte grafică (realizată ce-i drept cu ajutorul xml si javascript) dar care evidențiază în opinia mea noțiunea de *moștenire* în Python. Este vorba despre OpenERP, o aplicație modulară din sursă deschisă, de management al unei afaceri ce implică extrem de multe posibilități și utilizată din ce în ce mai mult de companiile interesate.

OpenERP se poate descărca de pe site-ul www.openerp.com și instala în mediile Windows (ușor) și Linux (Debian-Ubuntu, ceva mai greu dar nu foarte). Eu am încercat s-o instalez și pe Macintosh, însă după “lupte seculare” care au durat vreo două zile, m-am dat bătut...

OpenERP vine cu o serie de module standard preinstalate dar neactivate. Printr-un singur click însă ele se pot activa și apoi utiliza. Ceea ce însă este interesant e faptul că modulele se pot ajusta, se pot customiza, după dorința clientului. Poate că de exemplu unul dintre beneficiarii OpenERP vrea ca modulul “CRM” să conțină mai multe (sau mai puține) informații despre clienții lui din baza de date. Poate că ei vor ca partenerii de afaceri să nu aibă adresa sau numărul de telefon fix care să apară în rapoartele (chitanțe, facturi, etc) pe care programul le tipărește. Sunt o mulțime de posibilități sosite standard odată cu instalarea programului, dar care se pot ajusta pe gustul utilizatorului aplicației. Pentru a customiza însă aplicația veți fi nevoiți să pătrundeți în miezul (core-ul) ei și aceasta reprezintă deja un nivel de programare complex sau foarte complex.

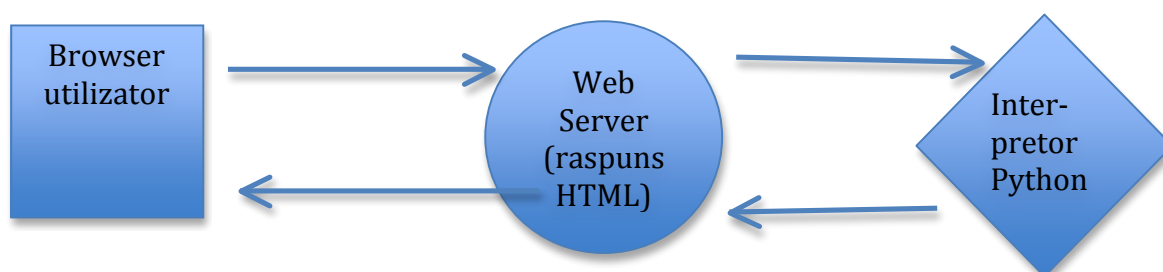
Moștenirea obiectelor înseamnă în esență mai puțină muncă pentru că veți prelua ceea ce alții au creat înainte și veți adapta – prin mici sau mai mari modificări, la nevoile dumneavoastră. Dacă aveți curiozitatea, puteți instala OpenERP (mai nou cunoscut ca Odoo) și apoi să-i deschideți fișierele diverselor module într-un editor de text. Veți putea analiza cum funcționează moștenirea în Python într-un program clădit pe o arhitectura

excelenta.

Cap. VIII Crearea si manipularea formularelor web

Daca ați avut răbdarea să parcurgeți această carte până aici, sunt sigur că v-ați întrebat la ce folosește totuși cunoașterea unui limbaj de programare. Ați vazut doar programe scrise într-un editor de text, rulate în interpretorul Python din Command Prompt (Windows) ori Terminal (MacOS sau Linux). Până acuma nu ați întâlnit un ecran prietenos cu o interfață grafică deosebită care să aibă în spate noianul de linii de cod scrise in Python. Și în general, în afară de *nerds*, nimeni nu utilizează calculatorul din linia de comandă.. Totuși Python deține capacitatea creării de interfețe grafice interesante (Graphical User Interface sau GUI). M-am gandit totuși să nu abordez (încă) acest subiect complex.

Python este însă un puternic limbaj de programare utilizat (și) pentru crearea aplicațiilor web. La fel ca formularele scrise în limbajul PHP, cele realizate în Python sunt găzduite pe un *webserver*, loc din care sunt servite utilizatorului interesat. Există însă o deosebire fundamentală între PHP și Python: în vreme ce scripturile PHP sălășluiesc numai și numai pe server pentru ca să poată fi procesate, scripturile Python pot să stea pe calculator atunci când avem de-a face cu o aplicație „desktop” de sine stătătoare și/sau pe web server atunci când e vorba de o aplicație pentru Internet. Cu PHP nu se pot face aplicații desktop³⁷. Simplist explicat, procesul cerere – procesare server - răspuns se desfășoară conform desenului de mai jos:



³⁷ Există totuși și posibilitatea rulării scripturilor PHP din Terminal cu comanda *php -a*, desigur dacă PHP este instalat pe calculatorul dumneavoastră.

Serverul web trebuie mai întâi configurat ca să recunoască scripturile (fișierele) Python care au – așa cum am învățat, extensia `.py`. Interpretorul Python procesează scriptul înainte de a-l trimite ca un răspuns HTML serverului, care la rândul lui îl returnează browser-ului utilizatorului. Un script Python cerut de browser poate să genereze un răspuns complet HTML descriind tipul conținutului în prima linie a fișierului astfel:

Content-type: text/html\r\n\r\n

Ce facem totuși cu scriptul (pe care încă nu l-am creat, linia de cod de mai sus arată doar declarația inițială din fișier), pentru că, altfel ca fișierele Python pe care le-am dezvoltat până acum și care rulau din linia de comandă a calculatorului, avem nevoie de un **web server** ?

Există aici două posibilități de luat în calcul:

- a. Să instalăm serverul arhicunoscut și arhifolosit³⁸ pe plan mondial Apache pe propriul computer și apoi să-l configurăm astfel încât să „știe” să proceseze scripturi Python.

Aceasta este o operație destul de dificilă, care necesită timp, aflată nu la îndemâna începătorilor. Totuși, în Anexe găsiți modul de configurare a fișierului `httpd.conf` (care este „creierul și inima” serverului Apache) pentru a prelucra fișiere Ruby, CGI, Perl și Python.

- b. Cea mai la îndemână metodă este să scriem propriul server în Python, operație ce necesită câteva linii de cod.

Vom apela la această a doua metodă pentru că este mult mai puțin complicată. Trebuie reținut că fișierele vor fi afișate în regim local (*localhost*) în browser-ul calculatorului dumneavoastră. Dacă aveți propriul website și vreți să exersați în „lumea reală,” puteți întreba administratorul de sistem al serviciului de găzduire (hosting) dacă există posibilitatea (în general, ea nu este setată implicit) de a vă configura serverul pentru scripturi Python. Oricum, dacă atunci când ați închiriat spațiu pe server în ofertă nu era trecută și opțiunea „scripturi CGI”, sigur nu aveți această posibilitate. Ca o paranteză vreau să mai adaug că găzduirea de scripturi sofisticate precum CGI, Perl, Python sau

³⁸ Peste 60% din website-uri utilizează serverul Apache

Ruby costă enorm – cel puțin la anumite firme și mai ales pentru buzunarele noastre. Serviciul meu de hosting – de care de altfel nu mă pot plânge, îmi oferea pentru scripturi Python un server „dedicat” la prețul „minim” de 29 euro/lună! Normal că am zis *pass* ca-ntr-o *clasă* Python! Există șansa să găsiți și oferte ce includ „scripturi CGI” la prețuri de bun simț, dar dacă spre exemplu, vă bate gândul să vă portați / alcătuiți un website în framework-ul Django (scris în Python) ori în Mezzanine, veți primi răspunsul că nu e posibil sau eventual, vă vor pretinde o căruță de bani.

Din acest motiv exemplele pe care le voi arăta în continuare sunt pur și simplu oferite în scop didactic, pentru a vedea cum poate fi utilizat Python în aplicații de Internet.

Dar vorba multă sărăcia programatorului, așa că să dăm drumul la treabă și să scriem codul serverului Python necesar aplicațiilor noastre, nu înainte de a crea pe desktopul computerului un folder numit „webdir” iar înăuntrul acestuia încă unul numit cgi-bin. Scriptul *webserver.py* arată astfel:

```

1 #!/usr/local/bin/python3
2
3 import os, sys
4 from http.server import HTTPServer, CGIHTTPRequestHandler
5 import cgitb; cgitb.enable()
6 webdir = '/Users/mirceaprodan/Desktop/webdir'
7 port = 8080
8
9 if len(sys.argv) > 1: webdir = sys.argv[1]
10 if len(sys.argv) > 2: port = int(sys.argv[2])
11 print('webdir "%s", port %s' %(webdir, port))
12
13 os.chdir(webdir)
14 srvraddr = ('', port)
15 srvrojb = HTTPServer(srvraddr, CGIHTTPRequestHandler)
16 srvrojb.serve_forever()

```

Fig. 8 1. Fișierul *webserver.py*

Urmăriți cu atenție liniile 1, 4 și 6. Ele sunt cele mai importante din întreg fișierul Python. Cu toate că prima linie este caracteristică computerelor UNIX (eu rulez serverul pe un computer Macintosh bazat pe UNIX) trebuie musai s-o adăugați în fișier chiar dacă rulați serverul pe Windows. Linia ar putea să difere, astfel încât să nu conțină „local” ci pur și simplu

```
#!/usr/bin/python3
```

Mai departe, scrieți cu atenție maximă linia 4, ea fiind cea mai importantă din întreg scriptul:

```
from http.server import HTTPServer, CGIHTTPRequestHandler
```

Apoi este linia 6 care arată locul exact unde se află fișierele Python ce urmează să fie servite în browser. Desigur că la dumneavoastră difera, mai cu seamă dacă lucrați în Windows.

Important este să scrieți cu exactitate unde se găsește folderul „webdir”:

```
webdir = '/Users/mirceaprodan/Desktop/webdir'
```

Linia 7 setează portul serverului. În mod normal, serverul web Apache „ascultă” la portul 80 și în cazul meu, acest lucru e valabil pentru că aici îl am și eu configurat pentru scripturi PHP . Atunci când scriu *localhost:80*³⁹ (sau *127.0.0.1:80*) îmi apare pagina principală a serverului (*It works!*) sau eventual (depinde cum e configurat) indexul serverului.

Așa cum vedeți în script, am setat serverul Python la valoarea 8080. Dacă aveți ceva împotriva numărului opt, puteți să-i dați oricare altă valoare în linia 7, dar neaparat mai mare ca 1024:

```
port = 8080
```

La sfârșit, după ce ați scris cu atenție toate liniile de cod exact așa cum sunt date în imaginea de mai sus, **salvați scriptul *webserver.py* în directorul (folderul) *webdir***.

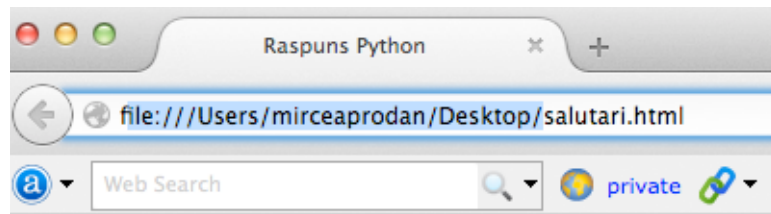
Nu știu dacă sunteți cât de cât familiarizați cu limbajul de marcare HTML, dar scriptul pe care îl vom rula inițial în browser este pur și simplu un document HTML (căruia mai târziu îi adăugăm funcția Python `print()` pentru a-l putea extrage de pe server). Să vedem mai întâi cum arată documentul HTML:

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Raspuns Python</title>
</head>
<body>
<h1>Salutari de la Python online!</h1>
</body>
</html>
```

Salavați-l ca *salutari.html* pe desktop și apoi dați dublu click pe el ca să-l rulați în

³⁹ 80 e implicit și de regulă nu se mai scrie

browser. Acesta este cum spuneam, doar un banal fișier HTML pe care îl putem rula direct de pe desktop, fără a fi necesar ca să fie încărcat în server, așa cum puteți vedea privind URL-ul evidențiat în browser:



Salutari de la Python online!

Fig. 8.2. *salutari.html* rulat în browser

Ca să avem un script Python adevărat trebuie doar să-i facem câteva retușuri, să-l încărcăm în server și apoi să-l afișăm în browser-ul web. Pentru asta trebuie mai întâi să adăugăm neapărat linia de cod care arată unde este interpretorul Python, inclusiv pentru fișierele pe care le rulați sub Windows:

```
#!/usr/local/bin/python3.3
```

Apoi dați un *Enter* ca să aveți o linie goală (linia 2) și după aceea scrieți linia de cod despre care am amintit la începutul acestui capitol:

```
print('Content-type:text/html\r\n\r\n')
```

În final, fiecare linie de program din fișierul *salutari.html* treceți-o drept argument al funcției `print()` din Python.

În consecință, prima noastră pagină web scrisă în Python (*raspuns.py*) va arăta așa:

```
1 #!/usr/local/bin/python3.3
2
3 print('Content-type:text/html\r\n\r\n')
4 print('<!DOCTYPE HTML>')
5 print('<html lang="en">')
6 print('<head>')
7 print('<meta charset="UTF-8">')
8 print('<title>Raspuns Python</title>')
9 print('</head>')
10 print('<body>')
11 print('<h1>Salutari de la Python online!</h1>')
12 print('</body>')
13 print('</html>')
```

Fig.8.3. Scriptul *raspuns.py*

Salvați fișierul *raspuns.py* în subfolderul *cgi-bin* aflat în folderul *webdir* situat pe desktop.

Acuma urmează lucrul cel mai interesant, anume vizualizarea primei dumneavoastră pagini web scrisă în Python. Pentru aceasta mai întâi trebui să pornim serverul despre care

am scris că lucrează la portul 8080.

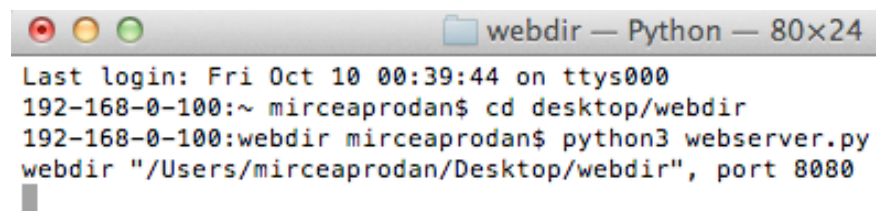
Schimbăm directorul în folderul *webdir* cu comanda (aceeași pentru Mac, Windows sau Linux) dată în Command Prompt sau Terminal :

```
cd Desktop/webdir
```

Următoarea comandă este:

```
python3 webserver.py
```

care pornește serverul la portul 8080 după cum se poate vedea în imaginea următoare:



```
webdir — Python — 80x24
Last login: Fri Oct 10 00:39:44 on ttys000
192-168-0-100:~ mirceaprodan$ cd desktop/webdir
192-168-0-100:webdir mirceaprodan$ python3 webserver.py
webdir "/Users/mirceaprodan/Desktop/webdir", port 8080
```

Fig.8.4. Pornire server

Deschidem browser-ul (daca nu este deja în această poziție) și scriem în bara URL:

<http://localhost:8080>

Dacă ați făcut corect toți pașii necesari, pagina web care apare va arăta indexul serverului ce include fișierele aflate în el. În cazul meu sunt mai multe, după cum se poate vedea mai jos:

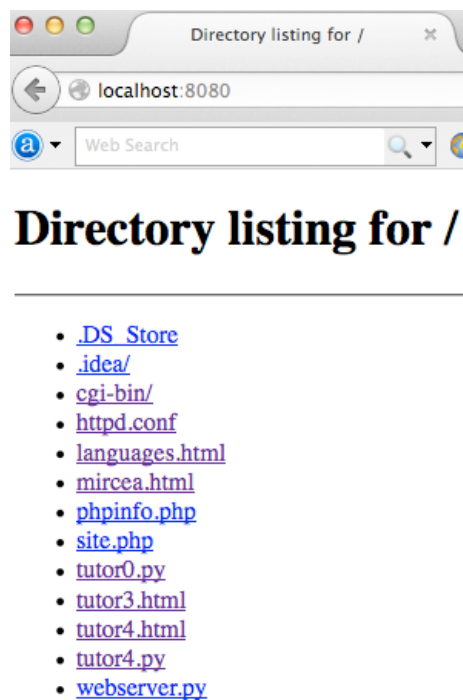
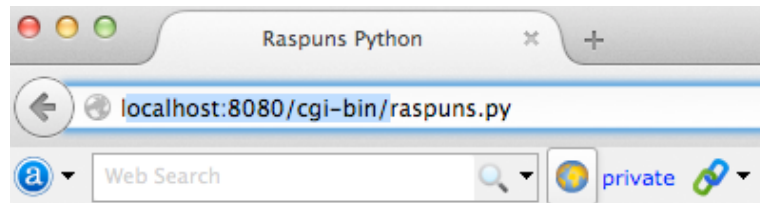


Fig.8.5. Directorul indexului fișierelor serverului

Cum scriam, fișierul care ne interesează și anume *raspuns.py* se află în folderul **cgi-bin**. Ca

să-l afișăm scriem în browser – în niciun caz nu dăm click în index pe subfolderul cgi-bin pentru ca va apărea o eroare de autorizare 403, următoarea adresă:

<http://localhost:8080/raspuns.py>:



Salutari de la Python online!

Fig.8.6. Scriptul web *raspuns.py* trimis din server

Se observă - privind URL-ul paginii, că de această dată fișierul este trimis din webserver și cu toate că pagina pare identică cu cea afișată de fișierul *raspuns.html*, de fapt nu este nici pe departe același lucru.

Următorul fișier pe care vi-l propun este mai complex și l-am preluat din cartea lui Mark Lutz, *Learning Python*. Este vorba despre un program foarte simpatic care afișează o pagină web ce conține codul salutului începătorilor în programare „Hello World!” afișat pentru cele mai cunoscute limbaje de programare. Fișierul Python este următorul (*limbaje.py*):

```
#!/usr/local/bin/python3.3

"""
show hello world syntax for input language name;
any languages name can arrive at this script since explicit
"""

debugme = False
inputkey = 'language'

hellos = {
    'Python': r" print('Hello World')",
    'Python2': r" print 'Hello Wordl'",
    'Perl': r' print "Hello World\n"',
    'Tcl': r' puts "Hello World"',
    'Scheme': r' (display "Hello World") (newline)',
    'SmallTalk': r" 'Hello World' print.",
    'Java': r' System.out.println("Hello World");',
    'C': r' printf("Hello World\n");',
    'C++': r' cout << "Hello World" << endl;',
    'Fortran': r" print *, 'Hello World'",
    'Pascal': r" Writeln('Hello World';)"
```

```

}

class dummy:
    def __init__(self, str): self.value = str

import cgi, sys
if debugme:
    form = {inputkey: dummy(sys.argv[1])}
else:
    form = cgi.FieldStorage()

print('Content-type: text/html\n')
print('<title>Languages</title>')
print('<h1>Sytnax</h1><hr>')

def showHello(form):
    choice = form[inputkey].value
    print('<h3>%s</h3><p><pre>' %choice)
    try:
        print(cgi.escape(hellos[choice]))
    except KeyError:
        print("Sorry-- I don'y know that language")
    print('</pre></p><br>')

if not inputkey in form or form[inputkey].value == 'All':
    for lang in hellos.keys():
        mock = {inputkey: dummy(lang)}
        showHello(mock)
else:
    showHello(mock)
print('</hr>')

```

Ieșirea lui este următoarea (partea superioară a paginii):

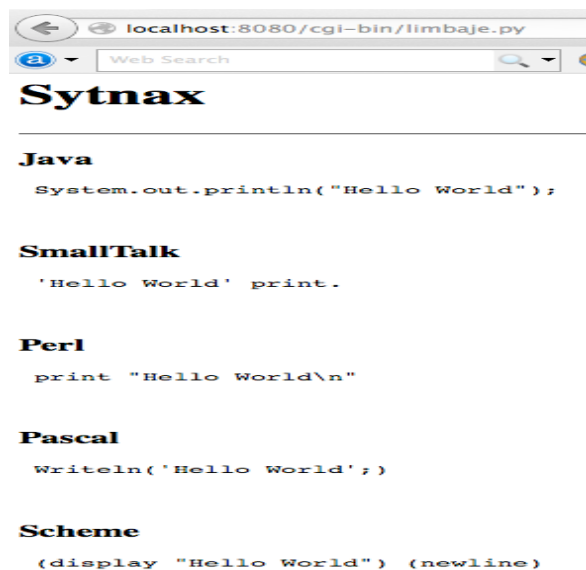


Fig.8.7. Ieșire program limbaje.py

8.1. Drepturi de acces la fișiere

Rămâne totuși de lămurit un lucru esențial în programarea Python pentru web, de care cu siguranță vă veți lovi și anume dreptul de acces la fișiere. Spre deosebire de scripturile PHP, fișierele Python sunt mult mai dificil de manipulat din server. Ce vreau să spun cu aceasta este că atunci când veți încerca să le afișați în browser, cu siguranță veți întâlni adesea refuzul serverului de a vă permite accesul la fișier sau, în cel mai bun caz, în loc să vă afișeze pagina web revendicată, vă va fi oferit fișierul în sine pe care să-l vizualizați sau descărcați ca în figura de mai jos (pentru MacOS dar e valabil și pentru Windows) unde eu am solicitat pagina web *link.py*:

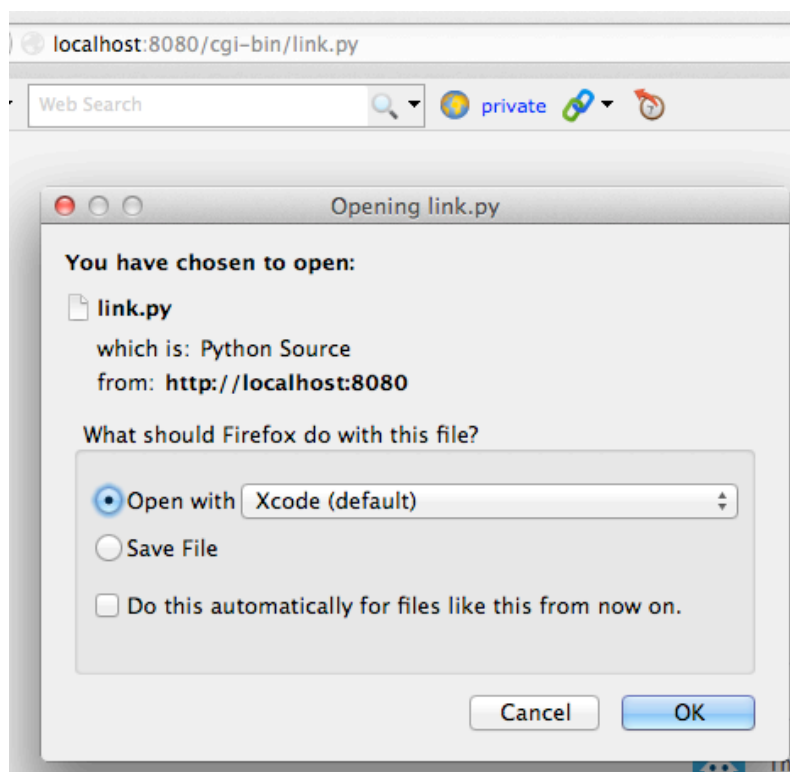


Fig.8.8. Setare incorectă a privilegiilor de acces

Pentru a evita o astfel de situație trebuie să configurați dreptul de acces la fișier astfel încât doar dumneavoastră ca administrator să fiți proprietarul lui și să aveți dreptul să scrieți în el și să-l citiți. Ceilalți, trebuie să aibă doar dreptul de a-l citi. Nu vă garantez ca această configurare funcționează fără probleme pentru că nu este așa. După ce am încercat de nenumărate ori diverse scripturi Python pentru web cu combinații diferite de privilegii, am

ajuns la concluzia că există o mare doză de impredictibilitate în a funcționa sau nu. Poate vi se pare o abordare neprofesională a problemei, dar după ce vă veți lovi de vreo 50 de ori de refuzul de a deschide pagina web cerută știind că ați scris/reprodus perfect un fișier, s-ar putea să îmi dați dreptate. Știu, în sistemele UNIX / Linux drepturile de acces la fișiere sunt mult mai restrictive, acesta fiind unul dintre motivele pentru care în general, utilizatorii comuni ai calculatoarelor evită folosirea lor. Nu știu cum este în Windows, nu am încercat, dar după știința mea, setarea privilegiilor asupra unui fișier nu este atât de spectaculoasă ca în Unix.

Milioane de posibilități...

Cum bine spunea unul dintre Părinții Fondatori ai Statelor Unite, cea mai bună metodă ca să înveți un lucru este să scrii o carte despre el. Pot spune că și eu am procedat la fel în cazul de față. Puteți afirma că nu e prea bine ca un individ care abia deslușește tainele unui limbaj de programare să se apuce să și scrie despre el. Munca mea la acest mic manual mi-a dovedit că nu este așa, pentru că mereu ești nevoit să cauți și să aduni informații de prin tot felul de cărți **în limba engleza** descărcate de pe Internet.

Din cărți se nasc alte cărți și această maximă e valabilă și în privința paginilor de față. Mi-au fost de un real ajutor astfel, volumele scrise de Tony Gaddis (din care m-am inspirat foarte mult) și Paul Barry & David Griffiths. Nu-i pot lăsa deoparte pe Dusty Phillips, pe Al Sweigart, ori pe Fabien Pinkaert, creatorul OpenERP.

Motivul pentru care m-am aplecat asupra acestui subiect este plăcerea de a scrie și rula programe în Python. Motivul cu adevărat important este că în limba română nu există până acum – după știința mea - o carte cât de neînsemnată despre acest din ce în ce mai important limbaj modern de programare. Nu am pretenția ca paginile mele să umple acest gol. Sunt convins că în România există destui oameni extrem de talentați în mărirea liniilor de cod Python – *cu siguranță autodidacți* - care ar putea s-o facă mult mai bine și cu mai mult talent ca mine. Sunt însă sigur că paginile pe care le-ați avut în față au reușit cât de cât să vă dăm o idee asupra unui minunat dar puțin cunoscut pe aceste meleaguri limbaj de programare.

Dacă v-am trezit curiozitatea, vă stau la dispoziție pe Internet o droaie de cărți, forumuri de discuții sau tutoriale dintre cele mai simple ori mai complexe. Dați doar un search pe motorul de cautare preferat și în față vi se vor deschide milioane de posibilități...

Bibliografie

4. Barry, Paul, Griffiths, David, *Head First Programming*, O'Reilly Media, Sebastopol, CA, 2009
5. Barry, Paul, *Head First Python*, O'Reilly Media, Sebastopol, CA, 2011
6. Gaddis, Tony, *Starting out with Python* (second edition), Pearson Education, Boston, 2012
7. Harrington, Andrew, *Hands-On Python. A tutorial introduction for beginners (Python 3.1. version)*, creativecommons.org
8. Phillips, David, *Python3 Object Oriented Programming*, Packt Publishing, London, 2010
9. Pinckaers Fabien, Van Vossel, Els, *Streamline your Manufacturing Processes with OpenERP*, Open Object Press, 2012
10. Swaroop, CH, *A byte of Python* (second edition)
11. Sweigart, Al, *Invent Your Own Computer Games With Python* (second edition), Creative Commons Licence