

ȘEDINȚA 4 – CLASA PYTHON PARTEA 2, EXCEPȚII.

Clasa în Python partea II

În următoare secțiune vom discuta despre metode statice. Aceasta metoda este foarte utilă când avem nevoie de o referință față de obiectele create.

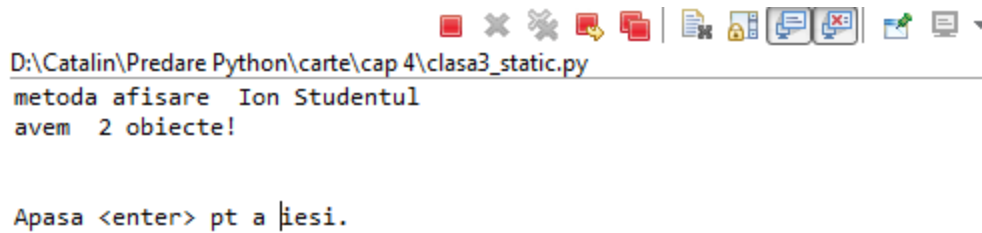
O metoda statica este o variabila creata in exteriorul metodelor; deci structura este:

```
>>> class NumeClasa(object):  
        """docstring"""  
        metoda_statica_1 = 0  
  
>>>
```

Fig.1

În Fig. 1 se poate vedea că am definit o variabilă numită metoda_statica_1 ce stochează valoarea 0. Această variabilă poartă numele de metoda statică. În cele ce urmează vom crea un program ce utilizează metoda statică pentru a explica mai bine acest concept.

```
# Program clasa3  
# Demonstreaza utilizarea static attributes  
# Ion Studentul - 1/26/13  
  
class NumeClasa(object):  
    """clasa mea"""  
    total=0  
    def __init__(self,nume):  
        """nume student"""  
        self.nume= nume  
        NumeClasa.total+=1  
  
    def metodaAfisare(self):  
        """afiseaza toate caracteristicile"""  
        print "metoda afisare\t", self.nume  
        print "avem ", NumeClasa.total , "obiecte!"  
  
obiect1=NumeClasa("Ion Studentul")  
obiect2=NumeClasa("Mihai Studentul")  
  
obiect1.metodaAfisare()  
  
raw_input("\n\nApasa <enter> pt a iesi.")
```



```
D:\Catalin\Predare Python\carte\cap 4\clasa3_static.py
metoda afisare Ion Studentul
avem 2 obiecte!

Apasa <enter> pt a iesi.
```

Fig. 2

Se poate vedea ca programul de mai sus definește un atribut în afara unei metode. Acest atribut poate fi accesat utilizand numele clasei prin apelarea `NumeClasa.total` (unde `total` este metoda statica definita in corpul clasei). Astfel programul propune numărarea obiectelor create.

Aceste attribute “statice” rămân neschimbate pe toata durata rularii, nu contează cate obiecte cream. Se poate vedea ca acest număr se incrementează cu fiecare obiect creat în metoda de inițializare `__init__` (`NumeClasa.total+=1`). Apoi de fiecare data când apelam metoda de afișare, programul ne va returna numărul stocat de metoda statica.

Am putea să cream pentru fiecare obiect acele attribute (si să le definim în `self`), dar ar deveni unice la nivel de obiect deoarece pentru fiecare obiect se va crea cate un atribut. Prin urmare nu am avea un atribut consistent peste toata clasa si nu am putea determina numarul de instante ale clasei (obiecte).

Apelarea unei metode statice nu se poate realiza fara numele clasei in fata.

```
>>> class NumeClasa(object):
    """clasa mea"""
    total=0
    def __init__(self,nume):
        """nume student"""
        self.nume= nume
        total+=1
```

```
>>> x=NumeClasa("Cata")
```

```
Traceback (most recent call last):
```

```
File "<pyshell#68>", line 1, in <module>
```

```
    x=NumeClasa("Cata")
```

```
File "<pyshell#67>", line 7, in __init__
```

```
    total+=1
```

```
UnboundLocalError: local variable 'total' referenced before assignment
```

Fig.3

În cele ce urmează putem vedea că apelarea unei metode statice se poate realiza și prin intermediul obiectelor, având același rezultat. Prin urmare, în Fig.4 putem vedea că `NumeClasa.total` returnează același număr ca și `x.total` sau `y.total`.

```
>>> class NumeClasa(object):
    """clasa mea"""
    total=0
    def __init__(self,nume):
        """nume student"""
        self.nume= nume
        NumeClasa.total+=1

>>> NumeClasa.total
0
>>> x =NumeClasa("Ion")
>>> NumeClasa.total
1
>>> x.total
1
>>> y=NumeClasa("Maria")
>>> NumeClasa.total
2
>>> y.total
2
>>> x.total
2
>>>
```

Fig.4

În exemplu următor vom crea o clasă ce are 4 obiecte. Acestea au un atribut unic numit `numarInstante`, creat la inițializarea obiectului, ce stochează valoarea standard de 0.

Apoi vom modifica acest atribut, pentru a arăta unicitatea fiecărui atribut raportat la un obiect.

```
# Program clasa5
# Demonstrează utilizarea metodei speciale __str__
# Ion Studentul - 1/26/13

class NumeClasa(object):
    """Clasa mea"""
    def __init__(self):
        self.numarInstante= 0

print ("cream 4 obiecte!")
obiect1=NumeClasa()
obiect2=NumeClasa()
obiect3=NumeClasa()
```

```

object4=NumeClasa()

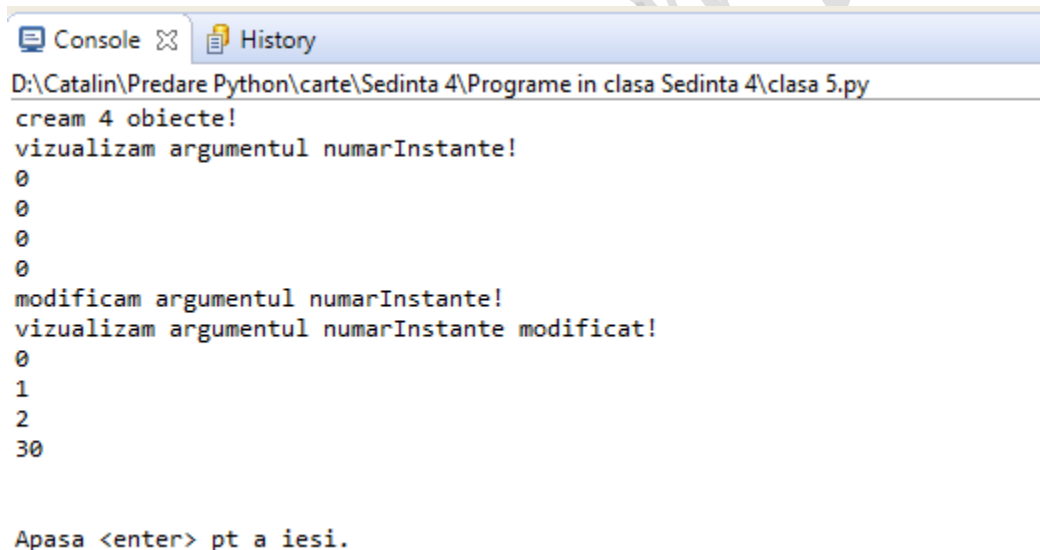
#vizualizam atributul numarInstante
print("vizualizam atributul numarInstante!")
print object1.numarInstante
print object2.numarInstante
print object3.numarInstante
print object4.numarInstante

print ("modificam atributul numarInstante!")
object1.numarInstante=0
object2.numarInstante=1
object3.numarInstante=2
object4.numarInstante=30

print("vizualizam atributul numarInstante modificat!")
print object1.numarInstante
print object2.numarInstante
print object3.numarInstante
print object4.numarInstante

raw_input("\n\nApasa <enter> pt a iesi.")

```



```

Console History
D:\Catalin\Predare Python\carte\Sedinta 4\Programe in clasa Sedinta 4\clasa 5.py
cream 4 obiecte!
vizualizam argumentul numarInstante!
0
0
0
0
modificam argumentul numarInstante!
vizualizam argumentul numarInstante modificat!
0
1
2
30

Apasa <enter> pt a iesi.

```

Fig. 5

In programul de mai sus vedem ca toate attributele numarInstante vor lua valoarea 0 la instantiere. Daca dorim sa modificam acest atribut acesta va avea valori diferite pentru fiecare obiect in parte. Retineti, fiecare obiect are un set unic de proprietati (attribute). Prin urmare nu vom avea o referinta fata de obiectele create.

Daca modificam programul de mai sus pentru a solicita metoda statica NumeClasa.numarInstante aceasta va genera eroare deoarece nu a fost definita. Astfel

in Fig.6 linia 34 a fost adaugata in plus fata de rularea anterioara. Se poate vedea ca, la rularea programului, interpretorul genereaza eroare.

```

28
29 print("vizualizam argumentul numarInstante modificat!")
30 print(object1.numarInstante)
31 print(object2.numarInstante)
32 print(object3.numarInstante)
33 print(object4.numarInstante)
34 print(NumeClasa.numarInstante)
35
36 raw_input("\n\nApasa <enter> pt a iesi.")
37

```

Console History

```

<terminated> D:\Catalin\Predare Python\carte\Sedinta 4\Programe in clasa Sedinta 4\clasa 5.py
1
2
30
Traceback (most recent call last):
  File "D:\Catalin\Predare Python\carte\Sedinta 4\Programe in clasa Sedinta 4\clasa 5.py", line 34, in <module>
    print NumeClasa.numarInstante
AttributeError: type object 'NumeClasa' has no attribute 'numarInstante'

```

In Fig. 7 schimbam valoarea stocata de total (metoda statica) la obiectele create. Vedem ca aceasta se updateaza, creandu-se attribute proprii ale obiectului x si y cu numele de total. Totusi metoda statica poate fi accesata inca prin intermediul clasei.

```

>>> class NumeClasa(object):
>>>     """clasa mea"""
>>>     total=0
>>>     def __init__(self,nume):
>>>         """nume student"""
>>>         self.nume= nume
>>>         NumeClasa.total+=1
>>>
>>> x =NumeClasa("Ion")
>>> y=NumeClasa("Maria")
>>> NumeClasa.total
2
>>> y.total = 20
>>> NumeClasa.total
2
>>> x.total=40
>>> NumeClasa.total
2
>>> x.total
40
>>> y.total
20
>>>

```

Fig.7

De asemenea, se poate să combinam obiectele create intr-un singur obiect sau putem să le alterăm cum dorim. Mai jos se poate observa un exemplu din aceasta practica:

```
# Program clasa6
# Demonstreaza utilizarea obiectelor
# Ion Studentul - 1/26/13

class NumeClasa(object):
    """clasa mea"""
    tipuriCarte = {"caro", "trefla", "inima rosie", "inima neagra"}
    numereCarte = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "A", "J", "Q", "K"}

    def __init__(self, tip, nr):
        """initializeaza variabile"""
        self.tip = tip
        self.nr = nr
        self.matchTip = 0
        self.matchNr = 0

    def __str__(self):
        """afiseaza toate caracteristicile"""
        if self.tip in NumeClasa.tipuriCarte:
            self.matchTip=1
        if self.nr in NumeClasa.numereCarte:
            self.matchNr=1
        if self.matchNr==1 and self.matchTip==1:
            ret=" "
        else:
            ret = " nu "

        return "Numarul si tipul ales de tine"+ret+"exista in packetul de carti!\n"
Valori:" + self.tip + ", " + self.nr

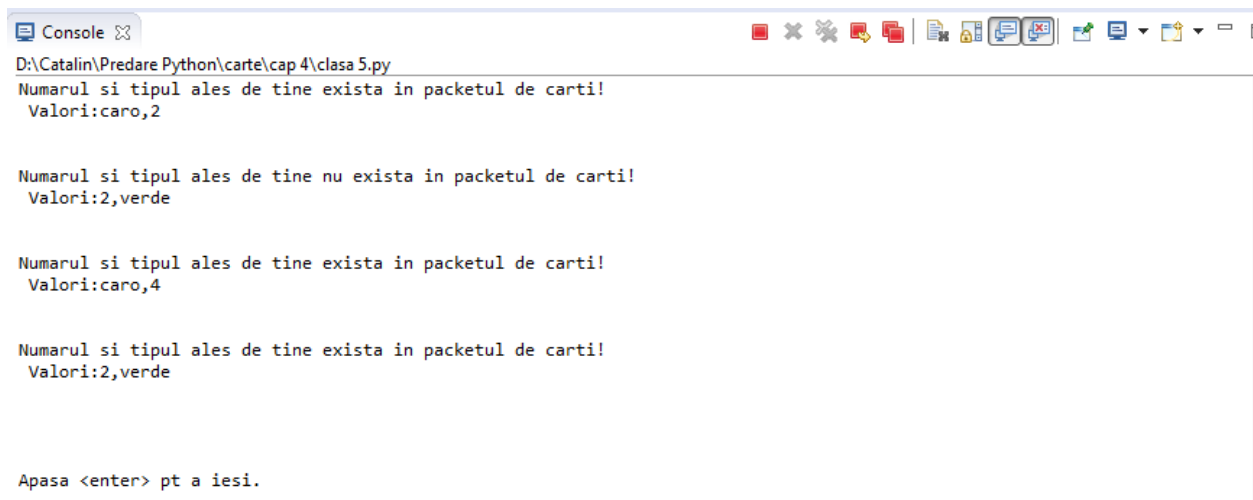
#cream primul obiect
obiect1=NumeClasa("caro", "2")
print obiect1
print "\n"

#cream al doilea obiect
obiect2=NumeClasa("2", "verde")
print obiect2
print "\n"

#alteram primul obiect
obiect1.nr="4"
print obiect1
print "\n"

#alteram al doilea obiect
obiect2.matchTip=1
obiect2.matchNr=1
print obiect2
print "\n"
```

```
raw_input("\n\nApasa <enter> pt a iesi.")
```



```

D:\Catalin\Predare Python\carte\cap 4\clasa 5.py
Numarul si tipul ales de tine exista in packetul de carti!
Valori:caro,2

Numarul si tipul ales de tine nu exista in packetul de carti!
Valori:2,verde

Numarul si tipul ales de tine exista in packetul de carti!
Valori:caro,4

Numarul si tipul ales de tine exista in packetul de carti!
Valori:2,verde

Apasa <enter> pt a iesi.

```

Fig. 8

În prima parte a programului cream o clasă în care definim două atribute în afara unei metode. Aceste atribute statice au rolul de a rămâne fixe pentru tot parcursul rulării și stochează două dicționare: un dicționar pentru tipul cărții ce poate exista într-un pachet de cărți (tipuriCarte) și un dicționar pentru numărul cărții ce poate exista în pachet (numereCarte).

Avem și două metode în această clasă.

Metoda utilizată la inițializarea unui obiect `__init__` ce inițializează cele două variabile în două atribute `self.tip` și `self.nr`. De asemenea, această metodă mai creează încă două atribute inițializate la zero (`self.matchTip = 0`, `self.matchNr = 0`) ce își prezintă utilitatea în metoda `__str__`.

În metoda `__str__` se realizează o verificare pentru valorile date la crearea obiectului; dacă acestea se regăsesc în dicționare. Dacă acest lucru se realizează atunci se returnează existența cărții în pachetul de cărți. În caz contrar va informa utilizatorul că nu există această carte. Prin urmare, această clasă realizează o verificare a atributelor de intrare ale unui obiect. Fiecare obiect creat este verificat la apelarea metodei `__str__`, metoda ce se apelează la printare obiectului.

Revenind la program, vom crea două obiecte: obiectul1 și obiectul2 pe care le vom printa pe rând să vedem rezultatul metodei `__str__`.

```
obiect1=NumeClasa("caro", "2")
```

```
print obiect1
obiect2=NumeClasa("2","verde")
print obiect2
```

Pentru primul obiect printare acestuia rezulta mesajul "Numarul și tipul ales de tine exista în packetul de carti! Valori:caro,2 " deoarece sirul de caractere <<caro>> a fost găsit în dicționarul NumeClasa.tipuriCarte, iar sirul de caractere <<2>> a fost găsit în dicționarul NumeClasa.numereCarte. Aceste răspunsuri pozitive vor determina ca attributele `self.matchTip` și `self.matchNr` să devina 1.

La printarea celui de al doilea obiect ne va returna un mesaj care indica ca nu este găsită cartea în pachetul de cărți datorita valorilor de intrare ale obiectului. Astfel când printam obiectul2 apelam metoda `__str__` ce va verifica dacă sirul de caractere <<2>> se afla în dicționarul NumeClasa.tipuriCarte, iar daca sirul de caractere <<verde>> se afla în dicționarul NumeClasa.numereCarte, ambele verificări vor rezulta răspunsuri negative deoarece la crearea obiectului am dat o ordine gresita, dar si datorita sirului de caractere <<verde>> ce nu se regaseste in dictionarul NumeClasa.tipuriCarte.

Aceste răspunsuri negative vor determina ca attributele `self.matchTip` și `self.matchNr` să rămână 0, valori ce vor determina răspunsul negativ și în mesajul returnat de metoda `__str__`. Mesajul este: " Numarul si tipul ales de tine nu exista în packetul de carti! Valori:2,verde"

În ultima partea a programului putem vedea o alterare a valorilor atributelor.

```
#altertam primul obiect
obiect1.nr="4"
print obiect1
```

La alterarea atributului `obiect1.nr` pentru primul obiect vedem ca se updateaza doar valoarea numărului cărții care nu schimba cu nimic mesajul returnat de metoda `__str__` așa cum putem vedea și în Fig. 8. Ambele siruri de caractere se regasesc în dictionarele date, deci mesajul este : "Numarul și tipul ales de tine exista în packetul de carti! Valori:caro,4 "

```
#alteram al doilea obiect
obiect2.matchTip=1
obiect2.matchNr=1
print obiect2
```

La alterarea atributelor celui de-al doilea obiect vedem ca se updateaza valorile atributelor `self.matchTip` și `self.matchNr` care vor genera o schimbare de mesaj returnata de metoda `__str__`. Astfel când printam obiectul2 apelam metoda `__str__` ce va verifica dacă 2 se afla în dicționarul NumeClasa.tipuriCarte, iar verde se afla în dicționarul NumeClasa.numereCarte, ambele verificări vor rezulta răspunsuri negative, ce nu vor modifica attributele `self.matchTip` și `self.matchNr`. Aceste attribute stochează deja valoarea 1, prin urmare vor genera mesajul: "Numarul si tipul ales de tine exista în packetul de carti! Valori:2,verde"

Cum am modifica programul pentru ca alterarea acestor attribute să nu poată induce mesaje false?

Iată o varianta mai jos:

```
# Program clasa6
# Demonstreaza utilizarea obiectelor
# Ion Studentul - 1/26/13

class NumeClasa(object):
    """clasa mea"""
    tipuriCarte = {"caro", "trefla", "inima rosie", "inima neagra"}
    numereCarte = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "A", "J", "Q", "K"}

    def __init__(self, tip, nr):
        """initializeaza variabile"""
        self.tip = tip
        self.nr = nr
        self.matchTip = 0
        self.matchNr = 0

    def __str__(self):
        """afiseaza toate caracteristicile"""
        #linii aditionale
        self.matchNr=0
        self.matchTip=0
        if self.tip in self.tipuriCarte:
            self.matchTip=1
        if self.nr in self.numereCarte:
            self.matchNr=1
        if self.matchNr==1 and self.matchTip==1:
            ret=" "
        else:
            ret = " nu "

        return "Numarul si tipul ales de tine"+ret+"exista in packetul de carti!/Valori:" + self.tip + ", " + self.nr

#cream primul obiect
obiect1=NumeClasa("caro", "2")
print obiect1
print "\n\n\n"
#cream al doilea obiect
obiect2=NumeClasa("2", "verde")
print obiect2
print "\n\n\n"
#altertam primul obiect
obiect1.nr="4"
print obiect1
print "\n\n\n"
```

```
#alteram al doilea obiect
obiect2.matchTip=1
print obiect2
print "\n\n\n"
```

```
raw_input("\n\nApasa <enter> pt a iesi.")
```

În programul de mai sus am recurs la o soluție ce presupune adăugarea a două linii în metoda `__str__`:

```
#linii aditionale
self.matchNr=0
self.matchTip=0
```

Aceste linii au rolul de a seta înainte de a rula restul codului metodei `__str__` atributele `matchNr` și `matchTip` la zero. Astfel, rescriem comportamentele setate manual.

O observație destul de pertinentă este că putem utiliza fie cuvântul cheie `self`, fie numele clasei pentru a putea apela un atribut sau o metodă din cadrul clasei. În exteriorul clasei putem utiliza fie cuvântul cheie `dat` de numele obiectului, fie numele clasei.

```
>>> class ClasaMea (object):
    x=1
    y=3
    def __init__(self):
        print self.x
        print ClasaMea.y

>>> print ClasaMea.y
3
>>> print obj1.y
3
>>>
```

Fig. 9

Nu există nici o diferență între apelarea atributului static fie prin numele clasei, fie printr-un obiect, cel puțin în acest exemplu.

Totuși să ne uităm la următoarea imagine pentru a vedea mici diferențe.

```

>>> class test(object):
        """clasa cu obiecte statice"""
        a = 1
        b = 1

>>> z=test()
>>> y=test()
>>> z.a
1
>>> y.a
1
>>> z.a= 10
>>> z.a
10
>>> y.a
1
>>> test.b=20
>>> z.b
20
>>> y.b
20
>>> test.a=40
>>> z.a
10
>>> y.a
40
>>>

```

Fig. 10

Creăm o clasă numită `test` ce are două metode statice `a = 1` și `b = 2`.

Apoi creăm două obiecte `z` și `y` ca fiind instanțe ale clasei.

Ambele obiecte au atributul `a` egal cu 1.

Voi altera atributul `a` al obiectului `z` să fie egal cu 10 și verific acest lucru. Verific dacă atributul `a` al obiectului `y` a rămas neschimbat. Vedem că `a` a rămas neschimbat datorită namespace-ului diferit al fiecărui obiect. Așa cum se regăsește specificat mai sus, un obiect poate avea un set unic de valori.

Dacă alterez atributul `b` la 20 apelat prin clasă vedem că aceste valori se vor regăsi la toate obiectele ce dețin acest atribut, deci și `y.b` și `z.b` vor avea valoarea 20.

Dacă aș altera un atribut care a fost modificat anterior ar trebui să fie modificat sau nu?

Prin urmare la început am modificat atributul `a` al obiectului `z` la 10. Dacă alterez atributul `a` la 40 apelat prin clasă dorim să vedem ce impact are asupra obiectului `z`.

Vedem ca z.a ramane neschimbat; totusi z.b isi va update valoarea. Prin urmare, valoarea data unui atribut prin obiect precede (are prioritate) valorii data unor atribute prin numele clasei.

Concluzie:

Setarea lui test.b la 20 duce la o rescriere a metodei statice b pentru ambele obiecte.

Setarea lui test.a la 40 duce la o rescriere a metodei statice a doar pentru obiectul y.

Obiectul z a ramas neschimbat deoarece valoarea data unui atribut prin obiect precede (are prioritate) valorii data unor atribute prin numele clasei.

Am învățat lucrurile cele mai elementare legate de clasele în Python. În următoarea parte vom învăța despre moștenire, recursivitate și comunicarea dintre clase.

În vederea studierii conceptului de recursivitate este necesara intelegerea namespaceului global sustinut intr-un dictionar. Globals este un dictionar ce mentine toate variabilele globale (din namespaceul global). Acesta se apeleaza prin globals(). Se poate vedea ca variabilele definite anterior precum x sau y se regasesc in dictionar.

```
>>> x= 1
>>> y=2
>>> z=4
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__package__': None, 'x': 1, 'y': 2, '__name__': '__main__', 'z': 4, '__doc__': None}
>>> .
```

Fig.11

Pentru a verifica daca o variabila exista putem folosi sintaxa:

```
if nume_variabila in globals():
```

Pentru a crea o variabila in dictionarul globals putem apela:

```
globals()[nume_variabila]=valoare stocata
```

Aceasta structura este intarita de Fig.12. Nu uitati ca trebuie sa treceti parantezele de la globals() pentru a putea functiona.

```

>>> globals()
{'__builtin__': <module '__builtin__' (built-in)>, '__name__':
'__main__', '__doc__': None, '__package__': None}
>>> a = "test"
>>> globals()
{'__builtin__': <module '__builtin__' (built-in)>, '__name__':
'__main__', '__doc__': None, 'a': 'test', '__package__': None}
>>> if "a" in globals():
    print "Este"

Este
>>> globals()["a"]
'test'
>>> globals["b"] = "test2"

Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    globals["b"] = "test2"
TypeError: 'builtin_function_or_method' object does not support
item assignment
>>>

```

Fig.12

Sa începem cu recursivitatea. Recursivitatea este procedeul de apelare din interiorul unei clase a aceleiasi clase de un număr de ori. Dar oare în Python clasa este recursiva? Se poate apela pe ea fără să dea eroare? Poate crea obiecte în interiorul clasei? Pentru toate acestea răspunsul este da, dar vom arata mai multe în exemplu ce urmează.

```

# Program clasa recursiva
# Demonstreaza recursivitatea clasei
# Ion Studentul - 1/26/13

class NumeClasa(object):
    """clasa mea"""
    instante=0
    def __init__(self):
        """initializarea obiectului"""
        NumeClasa.instante=NumeClasa.instante+1
    def creareinstante(self,nrInstante):
        """creaza multiple instante"""
        for i in range(nrInstante):
            globals()["x"+str(i)]=NumeClasa()

obiect1=NumeClasa()

obiect1.creareinstante(5)
print "Nr de instante existent este",obiect1.instante

print "Obiectul x3 are atributul instante ce este egal cu",x3.instante

raw_input("\n\nApasa <enter> pt a iesi.")

```

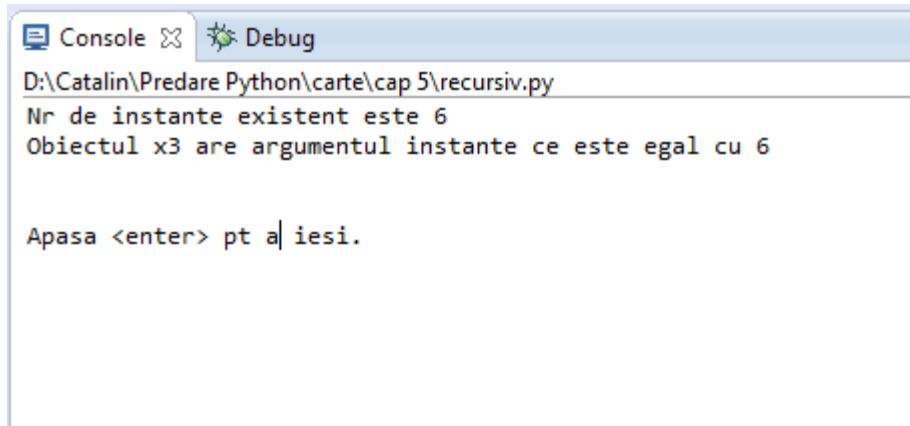


Fig. 13

În programul de mai sus se poate regăsi o clasă denumită `ClasaMea` care utilizează metoda de inițializare `__init__` ca și contor pentru instanțele create. Astfel de fiecare dată când un obiect este creat atunci contorul se incrementează (`self.instante`)

De asemenea mai avem o metodă denumită `creareInstante` care are rolul de a crea instanțe în mod automat. Aici regăsim un `for` care rulează de mai multe ori sintaxa:

```
globals()["x"+str(i)]=NumeClasa()
```

unde `i` este variabila care este ciclata și ia valori diferite.

`Global` este un dicționar unde sunt ținute toate variabilele programului declarate global. De aceea vom crea direct în acest dicționar acele variabile care pot fi accesate și în afara clasei. Dacă am fi dat doar `x` din păcate toate instanțele având același nume nu se făcea decât o rescriere a acelui obiect. Prin urmare nu se creau obiecte diferite.

Ba mai mult `x` nu putea fi accesat din exterior precum obiectul `x3`.

Programul de mai sus demonstrează și recursivitatea, deoarece clasa `ClasaMea` se apelează singură de câte ori este nevoie.

Dar oare nu putem apela o clasă din interiorul altei clase? Mai jos regăsim un program de tip joc ce realizează comunicarea între două clase. Clasele sunt ca un ecosistem în care toate elementele interacționează între ele.

```

# Program joc
# Demonstreaza apelarea unei clase in alta clasa
# Ion Studentul - 1/26/13

```

```

class Inamic(object):
    """Inamic"""
    def __init__(self):
        """initializarea inamicului"""
        self.viata=10

    def Ranire(self,rana):
        """Inamicul e ranit"""
        self.viata=self.viata-rana
        print "Viata inamicului a ajuns la", self.viata
        if (self.viata<=0):
            print "Inamicul a murit!"

class Erou(object):
    """Erou"""
    def __init__(self):
        """initializarea eroului"""
        self.viata=100
        self.sabie=2
    def Atac(self,rana,inamic):
        """Inamicul va fi ranit!"""
        inamic.Ranire(rana)

EroulMeu=Erou()

inamic1=Inamic()
print "\nIntalnim primul inamic!Deci il atacam cu sabia"
EroulMeu.Atac(EroulMeu.sabie,inamic1)

print "\nVom aplica inamicului un atac special"
EroulMeu.Atac(20,inamic1)

raw_input("\n\nApasa <enter> pt a iesi.")

```

```

D:\Catalin\Predare Python\carte\cap 5\Joc- comunicare intre doua clase.py

Intalnim primul inamic!Deci il atacam cu sabia
Viata inamicului a ajuns la 8

Vom aplica inamicului un atac special
Viata inamicului a ajuns la -12
Inamicul a murit!

Apasa <enter> pt a iesi.

```

Fig. 14

În programul de mai sus cream două clase. Prima clasă se numește Inamic.

Această clasă are două metode. În prima metodă observăm metoda de inițializare în care inițializăm atributul viață al inamicului la 10. Metoda Ranire primește o dată de intrare și anume rana. Trebuie să scădem această rană din viața totală a inamicului. Bineînțeles că verificăm dacă inamicul nostru a murit sau nu.

Cea de-a doua clasă se numește Erou și cu ajutorul ei putem avea un erou pe care să-l administram. Acesta are în componența două metode. Prima metodă este cea de inițializare, unde noi inițializăm două atribute ale eroului nostru. Prin urmare eroul nostru va avea viață setată la 100 și o sabie cu care poate răni inamicii făcând o vătămare de 2 unități din viața inamicului. Metoda Atac generează o rană inamicului ce este dată de intrare a acestei metode. Pentru o flexibilitate mai mare chiar și rana inamicului este dată de intrare.

Prima etapă a programului este crearea unui erou.

```
EroulMeu=Erou()
```

În a doua etapă întâlnim un inamic. Apelăm metoda atac pentru a răni inamicul.

```
EroulMeu.Atac(EroulMeu.sabie,inamic1)
```

În a treia etapă vom aplica inamicului un atac special ce va rezulta moartea inamicului conform Fig.2.

```
EroulMeu.Atac(20,inamic1)
```

Utilizarea moștenirilor în programarea OOP este un subiect foarte discutat. Acesta presupune crearea unei clase care moștenește toate proprietățile altei clase. Prin urmare se creează clase de tip părinte - copil (parent-child). Această ierarhie ne ajută să putem configura mult mai ușor programul nostru.

Să privim în exemplul de mai jos o moștenire de clase pentru a înțelege mai bine acest concept.


```
>>> class Unu(object):
    """prima clasa"""
    #argumente statice
    a = 1
    b = 2
    def __init__(self):
        """initializare arg"""
        self.c=3

>>> class Mosteneste_unu(Unu):
    """a doua clasa"""
    def __init__(self):
        """initializare arg"""
        self.x=25

>>> obj1=Unu()
>>> print obj1.a
1
>>> print obj1.b
2
>>> print obj1.c
3
>>> obj2=Mosteneste_unu()
>>> print obj2.a
1
>>> print obj2.b
2
>>> print obj2.x
25
```

Fig. 15

În Fig. 15 se poate observa că cream două clase. Prima clasă are nume sugestiv, `Unu()` și creează două atribute statice (`a=1` și `b=2`). În această clasă se poate vedea metoda de inițializare `__init__` care are ca rol crearea unui atribut `self.c`. Acesta este inițializat cu valoarea 3.

Vom crea și o a doua clasă numită sugestiv `Mosteneste_unu()` ce moștenește clasa `Unu` deoarece în loc de cuvântul cheie `object` pe care îl scriam până acum, vom scrie numele clasei pe care dorim să o moștenească, și anume `Unu`. Clasa `Mosteneste_unu()` apelează la crearea unui obiect metoda de inițializare `__init__()` ce are rolul de a crea un atribut numit `x` (`self.x=25`).

Vom crea primul obiect numit `obj1` prin apelarea clasei `Unu()`. După crearea obiectului vom afișa toate atributele acestui obiect.

Vom crea apoi un al doilea obiect numit `obj2` apelând clasa `Mosteneste_unu()`. Apoi vedem că putem afișa și atributele moștenite de la clasa `Unu()`. Spre exemplu putem afișa atributele `a` și `b` moștenite, dar și atributul `x` care este propriu-zis al clasei `Mosteneste_unu()`.

Dacă încercăm să printăm `obj1.x` ne va da eroare deoarece `obj1` nu are nici un atribut `x`: vezi figura 16.

Dar ce se întâmplă dacă încercăm să printăm `obj2.c`?

```
>>> print obj2.c

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    print obj2.c
AttributeError: 'Mosteneste_unu' object has no attribute 'c'
>>>
>>>
>>>
>>> print obj1.x

Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    print obj1.x
AttributeError: 'Unu' object has no attribute 'x'
```

Fig. 16

La printare `obj2.c` nu ar trebui să ne dea eroare dacă `Mosteneste_unu()` moștenește și metoda `__init__` din clasa `Unu()` și ar trebui să dea eroare dacă nu se moștenesc metodele. Dar clasa `Mosteneste_unu()` deja are o metoda `__init__`, prin urmare nu ar trebui să ne ridice o eroare compilatorul dacă metodele se moștenesc?

Pentru a putea explica ce se întâmplă cu variabila `c` trebuie să ne adresăm următoarele întrebări:

- Oare și metodele se moștenesc?
- Ce se întâmplă dacă și în clasa child și în clasa parent există aceleași atribute definite?
- Ce se întâmplă dacă și în clasa child și în clasa parent există aceleași metode?

Răspunsuri se pot regăsi mai jos:

Si metodele se moștenesc.

Dacă și în clasa child și în clasa parent există aceleași atribute definite, doar în clasa child vor fi rescrise valorile atributelor moștenite.

Dacă și în clasa child și în clasa parent există același metode, doar metodele definite în clasa child vor rescrie metodele moștenite din clasa parent.

O altă observație foarte importantă este că obiectele clasei parent nu se moștenesc de către clasa child. Nici obiectele clasei child nu se transmit către clasa parent. De fapt,

de la clasa child catre clasa parent nu exista nici o mostenire(deci clasa parent nu va lua nimic de la clasa child).

In cele ce urmează vom răspunde la fiecare întrebare adresata printr-un program exemplificator.

```
# Program mostenire -joc2
# Explica mostenirea
# Ion Studentul - 1/26/13

class Fiinta(object):
    """creaza o serie de proprietati ale unei fiinte"""
    def __init__(self):
        """proprietati mostenite de toate fiintele"""
        self.Reinviere()

    def Ranire(self,rana,fiinta_ranita):
        """fiinta eranita"""
        fiinta_ranita.viata=fiinta_ranita.viata-rana
        print "Viata a ajuns La", fiinta_ranita.viata, "(",fiinta_ranita.fiinta,")"
        if (fiinta_ranita.viata<=0):
            fiinta_ranita.viu=0
            print "Inamicul a murit!"
    def Reinviere(self):
        """reinvierea unei fiinte - reutilizarea inamicului sau a eroului"""
        self.viu=1
        self.viata=10
        self.sabie=2
        self.fiinta="inamic"

class Inamic(Fiinta):
    """Inamic"""
    pass

class Erou(Fiinta):
    """Erou"""
    def Proprietati_initiale(self):
        """initializarea eroului"""
        self.viata=100
        self.sabie=4
        self.fiinta="erou"

print '\tBine ati venit La jocul "Cavalerul"'
#cream eroul nostru
EroulMeu=Erou()
EroulMeu.Proprietati_initiale()
print "Iata proprietatile fiintei noastre(",EroulMeu.fiinta,"):"
print "viata :",EroulMeu.viata," puncte"
print "sabie :",EroulMeu.sabie," puncte vatamate din viata"

inamic1=Inamic()
print "\nIntalnim primul ",inamic1.fiinta,"!Deci il atacam cu sabia"
print "Iata proprietatile fiintei intalnite(",inamic1.fiinta,"):"
```

```

print "viata :", inamic1.viata, " puncte"
print "sabie :", inamic1.sabie, " puncte vatamate din viata"
EroulMeu.Ranire(EroulMeu.sabie, inamic1)

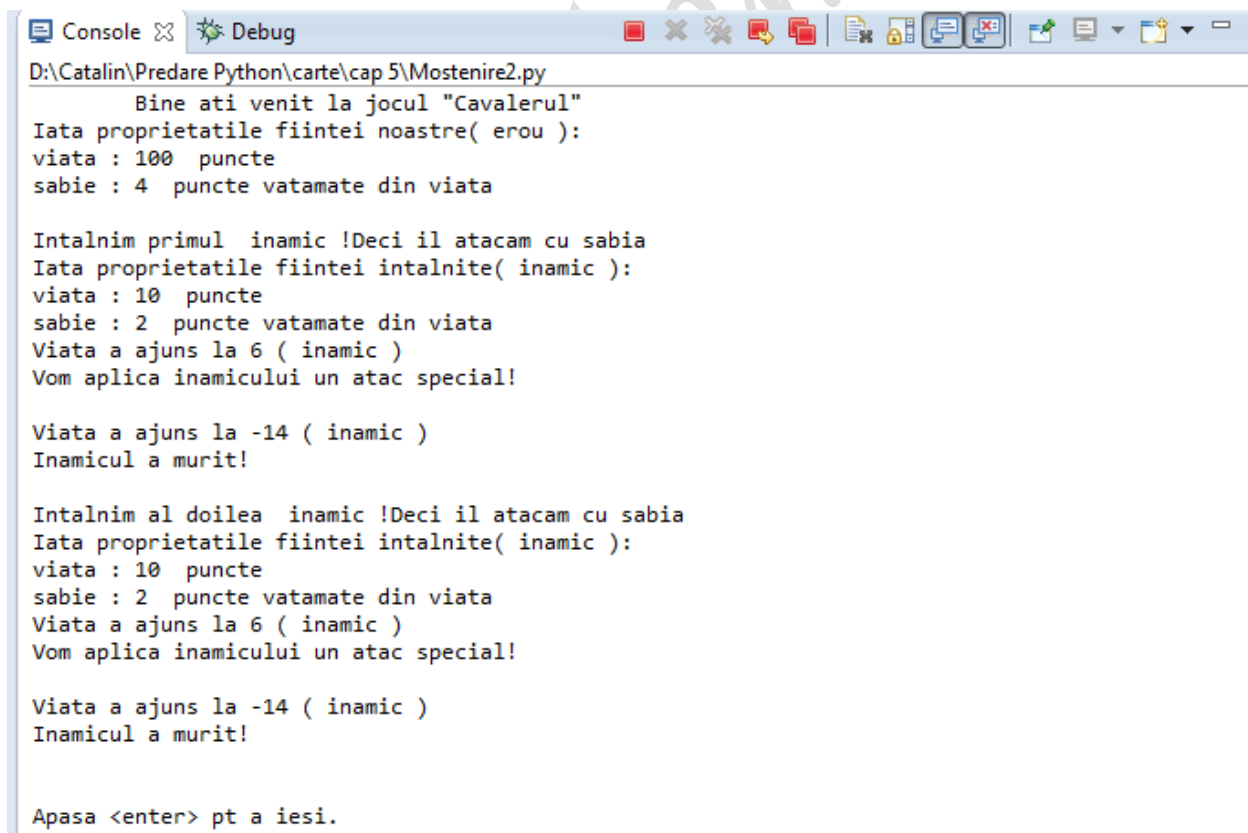
print "Vom aplica inamicului un atac special!\n"
EroulMeu.Ranire(20, inamic1)

inamic1.Reinviere()
print "\nIntalnim al doilea ", inamic1.fiinta, "!Deci il atacam cu sabia"
print "Iata proprietatile fiintei intalnite(", inamic1.fiinta, "):"
print "viata :", inamic1.viata, " puncte"
print "sabie :", inamic1.sabie, " puncte vatamate din viata"
EroulMeu.Ranire(EroulMeu.sabie, inamic1)

print "Vom aplica inamicului un atac special!\n"
EroulMeu.Ranire(20, inamic1)

raw_input("\n\nApasa <enter> pt a iesi.")

```



```

D:\Catalin\Predare Python\carte\cap 5\Mostenire2.py
Bine ati venit la jocul "Cavalerul"
Iata proprietatile fiintei noastre( erou ):
viata : 100 puncte
sabie : 4 puncte vatamate din viata

Intalnim primul inamic !Deci il atacam cu sabia
Iata proprietatile fiintei intalnite( inamic ):
viata : 10 puncte
sabie : 2 puncte vatamate din viata
Viata a ajuns la 6 ( inamic )
Vom aplica inamicului un atac special!

Viata a ajuns la -14 ( inamic )
Inamicul a murit!

Intalnim al doilea inamic !Deci il atacam cu sabia
Iata proprietatile fiintei intalnite( inamic ):
viata : 10 puncte
sabie : 2 puncte vatamate din viata
Viata a ajuns la 6 ( inamic )
Vom aplica inamicului un atac special!

Viata a ajuns la -14 ( inamic )
Inamicul a murit!

Apasa <enter> pt a iesi.

```

Fig. 17

În programul de mai sus începem prin a crea o clasă numită Fiinta(). Aceasta apelează în metoda de inițializare o altă metodă numită Reinviere(). Aceasta metodă are rolul de a defini proprietățile de bază ale obiectului de tip ființa și inițializează patru atribute la următoarele valori:

```
self.viu=1
self.viata=10
self.sabie=2
self.fiinta="inamic"
```

O a doua metodă este Ranire().

```
def Ranire(self, rana, fiinta_ranita):
    """fiinta eranita"""
```

Aceasta are doi parametri de intrare și anume, rana ce va avea rol decrementativ din viața ființei (atributul viată), dar și fiinta_ranita unde vom aplica acesta rana (scadere din viața).

Prin urmare vom scădea din viața ființei rana făcută, apoi vom afișa rezultatul doar cu scop vizual.

```
fiinta_ranita.viata=fiinta_ranita.viata-rana
print "Viata a ajuns la", fiinta_ranita.viata, "(", fiinta_ranita.fiinta, ")"
```

În următorul pas vom face verificări să vedem dacă ființa ranița a ajuns la capătul vieții sau nu:

```
if (fiinta_ranita.viata<=0):
    fiinta_ranita.viu=0
    print "Inamicul a murit!"
```

O a doua clasă este Inamic(). Aceasta clasă moștenește clasa Fiinta.

```
class Inamic(Fiinta):
    """Inamic"""
    pass
```

Prin urmare apelăm în interiorul clasei cuvântul cheie pass deoarece nu dorim să adăugăm / modificăm nimic. Dorim să arătăm ce se întâmplă în cazul în care o clasă care moștenește o altă clasă, dacă va moșteni și toate atributele și metodele definite de clasa părinte. În unele cazuri clasa care va fi moștenită se numește parent (părinte), iar cea care moștenește parent va fi numită child (copil). În cazul nostru parent este Fiinta(), iar child este Inamic().

Creăm și o clasă numită Erou() care și ea, la rândul ei, moștenește clasa Fiinta() cu toate metodele și atributele ei.

```
class Erou(Fiinta):
```

Datorita moștenirii dacă cream o metoda cu același nume am rescrie aceea metoda pentru clasa child. Deci dacă dorim să alterăm valorile moștenite trebuie să cream o altă metoda diferită de cea de inițializare. Dacă cream o metoda de inițializare vom rescrie metoda de inițializare moștenită de la clasa parent-Fiinta(). Cream metoda `proprietati_initiale()` ca fiind clasa ce trebuie apelată pentru alterarea de valori moștenite. Aceasta metoda alterează atributele viața, sabie și ființa pentru ca eroul nostru să aibă o viață mai rezistentă rănilor provocate de inamic, să aibă o sabie care renaște mai mult și cel mai important să alterăm ființa ca fiind erou.

```
def Proprietati_initiale(self):
    """inițializarea eroului"""
    self.viata=100
    self.sabie=4
    self.fiinta="erou"
```

Aceste valori le putem modifica să cream stagii de dificultate în ceea ce privește jocul nostru, și aici mă refer la o sabie ce poate răni mai puțin sau mai mult sau la o viață care rezistă la mai multe sau mai puține atacuri venite din partea inamicului.

Trebuie să reținem ca, afirmând ipotetic, dacă o clasa are 10 metode, iar aceasta clasa devine o clasa părinte, aceste 10 metode distincte se moștenesc în întregime de către clasa copil. Dacă în clasa părinte și în clasa copil există același atribut sau aceleași metode, la crearea unui obiect apelând clasa copil vom avea atributele sau metodele suprascrise de aceasta dublură.

În următoarea secțiune a programului vom afișa o linie care ne arată ca noul nostru script dorește să devină un joc, apoi cream un obiect numit sugestiv intitulat EroulMeu prin apelarea clasei Erou(). Aceasta clasa moștenește clasa Fiinta(), ce conține o metoda de inițializare. Așa cum am prezentat mai sus, metoda de inițializare din clasa Fiinta apelează metoda Reinviere(), ce va inițializa cele patru atribute.

```
print '\tBine ati venit la jocul "Cavalerul"'
#cream eroul nostru
EroulMeu=Erou()
```

Dacă dorim să modificăm atributele inițializate indirect de metoda Reinviere(), vom apela metoda `Proprietati_initiale()`.

```
EroulMeu.Proprietati_initiale()
```

În următoarea secțiune vom afișa proprietățile eroului nostru. Pe lângă informarea jucătorului, aceste afișări au și rol didactic, ca noi să putem vedea ca suprascrierea atributelor moștenite a avut loc.

```
print "Iata proprietatile fiintei noastre("",EroulMeu.fiinta,""):"
print "viata :",EroulMeu.viata," puncte"
```

```
print "sabie :",EroulMeu.sabie," puncte vatamate din viata"
```

Un următor pas al acestui program este întâlnirea cu un inamic. Prin urmare, mai întâi cream un inamic prin crearea unui obiect inamic1 apelând clasa Inamic().

```
inamic1=Inamic()
```

Vom afișa aceasta întâlnire cu inamicul, unde se poate observa ca valoarea a atributului fiinta este dat de clasa parent, valoare nealterata de noi.

```
print "\nIntalnim primul ",inamic1.fiinta,"!Deci îl atacam cu sabia"
```

În următoarea secțiune vom afișa proprietățile eroului nostru. Cum spuneam, aceste afișări au și rol didactic, ca noi să putem vedea ca suprascrierea atributelor moștenite a avut loc.

```
print "Iata proprietatile fiintei intalnite(",inamic1.fiinta,"):"
print "viata :",inamic1.viata," puncte"
print "sabie :",inamic1.sabie," puncte vatamate din viata"
```

Din nou putem vizualiza valorile nealterate ale atributelor moștenite de la clasa părinte. În scopul de a apărea eroul de inamic va trebui să-l atacăm cu sabia. Astfel apelăm metoda moștenită de la clasa parent Ranire(), prin care îi pricinuim o rană unui inamic ce-l dam ca parametru de intrare al metodei Ranire.

```
EroulMeu.Ranire(EroulMeu.sabie,inamic1)
```

Bineînțeles ca exista diverse abordări ale acestei probleme. Puteam să apelăm metoda Ranire() de la obiectul inamic1. Astfel nu mai era necesar să dam ca parametru de intrare ființa ranită. Totuși pentru a demonstra flexibilitatea mare pe care o aveți în Python am ales această abordare. Putem vedea ca parametrii unei metode sau funcții pot fi clar și obiecte. Astfel putem să prelucram acel obiect precum dorim în clase sau funcții diferite.

Pentru a omori inamicul vom aplica o valoare mai mare decât viața lui printr-un atac special.

```
print "Vom aplica inamicului un atac special!\n"
EroulMeu.Ranire(20,inamic1)
```

Am putea crea câte un obiect pentru fiecare inamic întâlnit, dar ar fi o abordare ce consuma resurse și memorie. Mai bine ar fi dacă am reînvia un inamic și l-am refolosi de câte ori ne dorim acest lucru.

```
inamic1.Reinviere()
```

Restul codului de mai sus este duplicat pentru a putea arata ca un inamic ce este reînviat poate fi folosit la fel.

Sa analizam un pic programul din alta perspectiva. Vedem ca atributul `viața` poate lua valori negative. Bineînțeles ca acest comportament se poate îndrepta deoarece avem o metoda care se ocupa cu ranirea ființei.

Sa elaboram un pic idea și să vizualizam ce avem în momentul de fata în programul precedent pentru a modifica atributul `viața` al unui obiect de tip `fiinta`:

```
EroulMeu.Ranire(EroulMeu.sabie,inamic1)
```

Daca în cadrul corpului principal al programului am fi scris în loc de fraza de mai sus ar fi avut același rezultat:

```
inamic1.viața=inamic.viața-EroulMeu.sabie
```

Totuși trebuie să înțelegem ca folosirea de metode duce la posibilitatea de a aduce multe beneficii cum ar fi verificarea și modificarea complexa a elementelor. De asemenea, un al doilea motiv pentru care folosim metode înlocuind operațiile directe este scalabilitatea. Dacă în loc de aceste operații vom avea o metoda, de fiecare data când modificam metoda trebuie să facem aceasta operație dintr-un singur loc.

Aceste sunt de fapt motivele pentru care utilizam metode pentru orice operație pe care dorim să o realizăm.

Un alt aspect ce trebuie discutat este încapsularea obiectelor (eng. object encapsulation). Deci similar cu variabilele definite în cadrul unei funcții care sunt semnificativ local, fără a putea fi accesate direct așa se întâmplă și cu atributele unui obiect. Nu pot fi accesate direct ci doar prin acel obiect.

Totuși, toate atributele unui obiect sau metodele unei clase pot fi clasificate în publice și private. Toate atributele și metodele sunt standard clasificate ca fiind publice, adică se pot accesa prin intermediul acelui obiect sau invocate cu ajutorul clasei.

Pentru a încuraja încapsularea, în Python se poate regăsi și crea atribute și metode private, ceea ce înseamnă ca accesarea din exterior nu este permisă în mod direct pentru acel element privat (metoda sau atribut).

Sa vedem un exemplu elocvent în ceea ce privește atributul și metoda privata.

O metoda privata se creează prin utilizarea a doua caractere underline `__` în fata numelui metodei, iar un atribut privat se creează folosind tot doua caractere underline `__` în fata numelui metodei.


```

Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>>
class Clasa_privat(object):
    """test clasa privata"""
    def test_public(self):
        """o metoda publica cu un argument privat"""
        self.argumentPublic = "public"
        self.__argumentPrivat = "privat"

    def acceseazaArgPrivat(self):
        """o metoda publica ce acceseaza un argument privat"""
        print self.__argumentPrivat," <= argument privat "

    def __MetodaPrivata(self):
        """o metoda privata"""
        print self.__argumentPrivat," <= argument privat "
        print self.argumentPublic," <= argument public"

    def acceseazaMetodaPrivata(self):
        """o metoda publica ce acceseaza o metoda privata"""
        print self.__argumentPrivat," <= argument privat "

>>> obj1=Clasa_privat()

```

Fig. 18

În Fig.18 se poate vedea că creăm o clasă numită `Clasa_privat` care conține 4 metode.

Prima metodă `test_public()` este o metodă care definește un atribut public (`self.argumentPublic`) și un atribut privat (`self.__argumentPrivat`).

A doua metodă `acceseazaArgPrivat()` accesează atributul privat pentru a se observa că nu ridică nici o eroare dacă un atribut privat este accesat prin intermediul unei metode.

A treia metodă este o metodă privată numită `__MetodaPrivata()`. Această metodă privată utilizează două atribute cu scopul de a demonstra că o metodă privată nu trebuie să folosească neapărat atribute private.

A patra metodă este metoda `acceseazaMetodaPrivata()` care are ca scop de a utiliza metoda privată creată anterior pentru a se observa că o metodă privată utilizată în cadrul altei metode nu ridică nici o eroare.

Vom crea un obiect al acestei clase numit `obj1`.

```
>>> obj1.test_public()
>>> obj1.__argumentPrivat

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    obj1.__argumentPrivat
AttributeError: 'Clasa_privat' object has no attribute '__argumentPrivat'
>>> obj1.argumentPrivat

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    obj1.argumentPrivat
AttributeError: 'Clasa_privat' object has no attribute 'argumentPrivat'
>>>
```

Fig. 19

Vom apela metoda `test_public()` pentru a crea un atribut public (`self.atributPublic`) și un atribut privat (`self.__atributPrivat`).

Apoi vom încerca să apelăm atributul privat. Vom vedea ca ridică eroare fie dacă îl apelăm în formă cu underscore sau nu.

```
>>>
>>> obj1.accesseazaArgPrivat()
privat <= argument privat
>>>
```

Fig. 20

Dacă vom apela metoda care utilizează atributul privat nu vom avea nici o eroare, așa cum indică și Fig.20. Totuși cei ce au proiectat Python-ul au adăugat și o “portită”, o utilizare prin care putem apela prin intermediul obiectului fără a mai folosi o metoda, cum se poate vedea și în Fig.21.

```
>>> obj1._Clasa_privat__argumentPrivat
'privat'
>>> |
```

Fig. 21

Pentru a apela prin intermediul obiectului trebuie să folosim underscore numeClasa underscore underscore AtributPrivat.

De asemenea același comportament de eroare apare și la apelarea directă a unei metode private așa cum indică Fig.22.

```
>>> obj1.__MetodaPrivata()

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    obj1.__MetodaPrivata()
AttributeError: 'Clasa_privat' object has no attribute '__MetodaPrivata'
>>> obj1.MetodaPrivata()

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    obj1.MetodaPrivata()
AttributeError: 'Clasa_privat' object has no attribute 'MetodaPrivata'
>>>
```

Fig. 22

Daca vom apela metoda privata prin intermediul unei metode publice nu va ridica nici o eroare. De asemenea aceeași utilizare prin care putem apela prin intermediul obiectului un atribut privat funcționează și la o metoda privata., cum se poate vedea și în Fig.23.

```
...
>>> obj1.accesseazaMetodaPrivata()
privat <= argument privat
>>> obj1.__Clasa_privat__MetodaPrivata()
privat <= argument privat
public <= argument public
>>> |
```

Fig. 23

Trebuie sa facem o observatie referitoare la crearea unei clase. Prin scrierea cuvântului object între parantezele unei clase la definire noi realizăm o mostenire.

Polimorfism

Polimorfismul este calitatea ce permite să utilizezi același tip de funcție sau clasa cu tipuri diferite. Am văzut de multe ori exemple polimorfe. Unul din ele este funcția len() ce poate fi utilizată cu tipuri diferite de variabile sau date (sir de caractere, lista, tuplu), cum ar fi în exemplu de mai jos:

```
>>> len("Cat ani crezi ca am?")
20
>>> len((1, 2, 3, 4, 5))
5
>>> len(["a", "b", "c"])
3
```

Utilizat în contextul de OOP, polimorfism înseamnă ca puteți trimite același mesaj către obiecte din diferite clase legate între ele prin moștenire și de a obține rezultate diferite.

```

class Animal:
    def __init__(self, name):      # metoda de initializare a clasei
        self.name = name
    def vorbeste(self):           # Metoda abstractacta
        pass

class Pisica(Animal):
    def vorbeste (self):
        return 'Miau!'

class Caine(Animal):
    def vorbeste (self):
        return 'Ham!'

obj1 = Pisica ('Lola')
obj1.vorbeste()
obj2 = Caine ('Lassie')
obj2.vorbeste()

```

```

Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> class Animal:
    def __init__(self, name):      # metoda de initializare a clasei
        self.name = name
    def vorbeste(self):           # Metoda abstractacta
        pass

>>> class Pisica(Animal):
    def vorbeste (self):
        return 'Miau!'

>>> class Caine(Animal):
    def vorbeste (self):
        return 'Ham!'

>>> obj1 = Pisica ('Lola')
>>> obj1.vorbeste()
'Miau!'
>>> obj2 = Caine ('Lassie')
>>> obj2.vorbeste()
'Ham!'
>>> |

```

Fig. 24

Se poate observa ca toate animalele pot vorbi dar vorbesc diferit. Deci comportamentul vorbeste al unui animal devine polimorfic în sensul ca este realizat diferit în funcție de animal.

Lucrul cu fişierele

Fişierele sunt categorisite ca fiind fie text, fie de tip binar.

Cu Python este foarte uşor să citeşti fişiere de tip text (eng. plain text)- fişiere ce sunt create doar cu caractere ASCII. Prin urmare, un fişier text nu înseamnă neapărat un fişier cu extensia .txt, ci toate fişierele care nu conţin caractere diferite de cele ASCII.

Fişierele plain text sunt o soluţie excelentă pentru stocarea de informaţii deoarece sunt independente de sistemul de operare.

Un fişier text este structurat în linii, iar o linie este o secvenţă de caractere ce are o serie de caractere cu scop delimitator la final numit EOL (end of line). Fiecare sistem de operare are un caracter de tip EOL diferit. Spre exemplificare, Linux foloseşte “\n” ca şi EOL, iar Windows foloseşte fie “\n\r”, fie “\n”.

Un fişier binar este un fişier ce conţine şi alte elemente non-ASCII. Fişierul binar poate fi procesat doar de aplicaţia ce cunoaşte structura acelui fişier.

Pentru a deschide un fişier trebuie să folosim funcţia open(). Open returnează un obiect şi trebuie să dam la apelare două atribute.

Sintaxa este :

fişier_object = open(umeFisier, mod) unde fişier_object este variabila care susţine obiectul fişier returnat. Atributul umeFisier trebuie să conţină şi extensia şi este de tip şir de caractere. De asemenea şirul de caractere umeFisier poate să fie o cale absolută sau relativă.

Calea absolută este calea prin sistemul de fişiere de la rădăcina sau drive până la fişierul dorit. Calea relativă se raportează la directorul curent unde acel program rulează.

Atributul mod este de tip şir de caractere şi este opţional; dacă îl omitem se va considera ca fiind ‘r’.

Modurile pot fi:

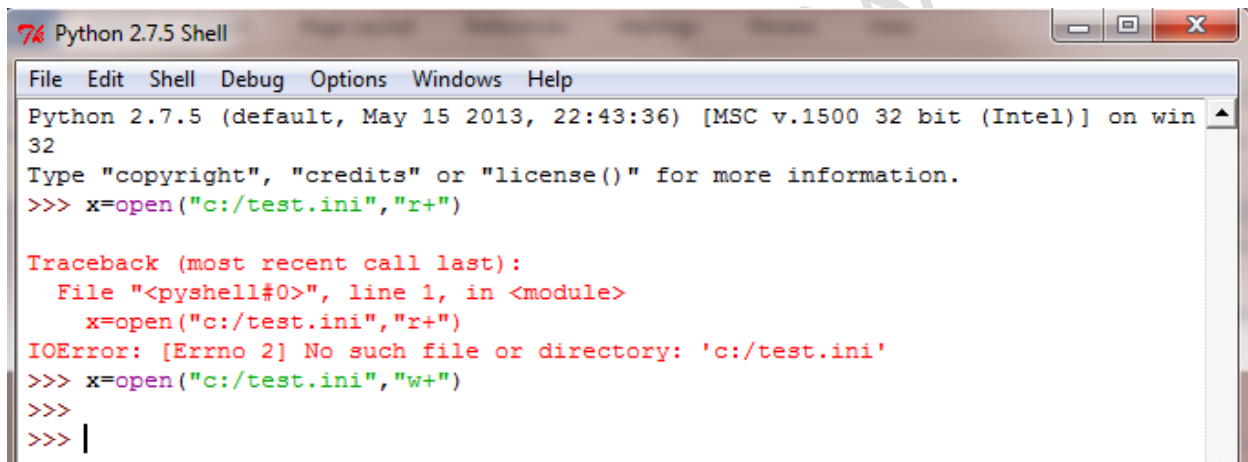
- ‘r’ când vrem să citim din fişier (exclusiv citire). Fişierul trebuie să existe!
- ‘w’ când dorim să scriem în fişier (exclusiv scriere). Dacă fişierul nu există va fi creat. Dacă fişierul deja există conţinutul vechiului fişier va fi şters.
- ‘a’ deschide fişierul pentru scriere(exclusiv scriere). Dacă fişierul nu există va fi creat. Orice date scrise cu a va fi adăugate la final.
- ‘r+’ când vrem să citim şi să scriem fişierul. Fişierul trebuie să existe!
- ‘w+’ când vrem să citim şi să scriem fişierul. Dacă fişierul nu există va fi creat. Dacă fişierul deja există conţinutul vechiului fişier va fi şters.
- ‘a+’ când vrem să citim şi să scriem fişierul. Dacă fişierul nu există va fi creat. Orice date scrise cu ,a’ va fi adăugată la final.

Mai jos (Fig.42) putem vedea un exemplu în care deschidem un fișier printr-o cale absolută pentru citire.

```
>>> f=open("C:/test.ini","r")
>>> print f
<open file 'C:/test.ini', mode 'r' at 0x02359D88>
>>> |
```

Fig. 25

Crearea unui fișier se face utilizând funcția open, dar modificând modul folosit. Fig. 26 exemplifica utilizarea modului r+ care poate face citire și scriere la un fișier inexistent, practica ce va ridica o eroare. De asemenea, pentru același fișier vom folosi 'w+' atunci va crea fișierul dorit fără a ridica eroare.



```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> x=open("c:/test.ini","r+")

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    x=open("c:/test.ini","r+")
IOError: [Errno 2] No such file or directory: 'c:/test.ini'
>>> x=open("c:/test.ini","w+")
>>>
>>> |
```

Fig. 26

Citirea unui fișier se poate face prin multiple metode, și anume 4 soluții:

file.read() –aceasta metoda de citire va returna tot conținutul fișierului într-un singur șir de caractere.

file.readline() – aceasta metoda de citire citește linie cu linie. De fiecare data când este apelata va returna o linie.

file.readlines() – aceasta metoda citește toate liniile, iar fiecare linie este un element dintr-o lista.

A patra soluție este să facem o buclare a fișierului. Mai jos regăsim un exemplu:

```
file = open('newfile.txt', 'r')
for line in file:
```

```
print line,
```

O să cream un fișier numit text_importat1 cu extensia .txt ce va conține următoarele trei linii ce va fi salvat în același director cu programul .py de mai jos:

Linia1.

Linia2 fisier= text.

Linia 3 nume= text_importat1

Vom crea un program ce are ca scop vizualizarea informațiilor acumulate în ceea ce privește citirea unui fișier.

```
# Program Private methonds and atributs
# Explica mostenirea
# Ion Studentul - 1/26/13

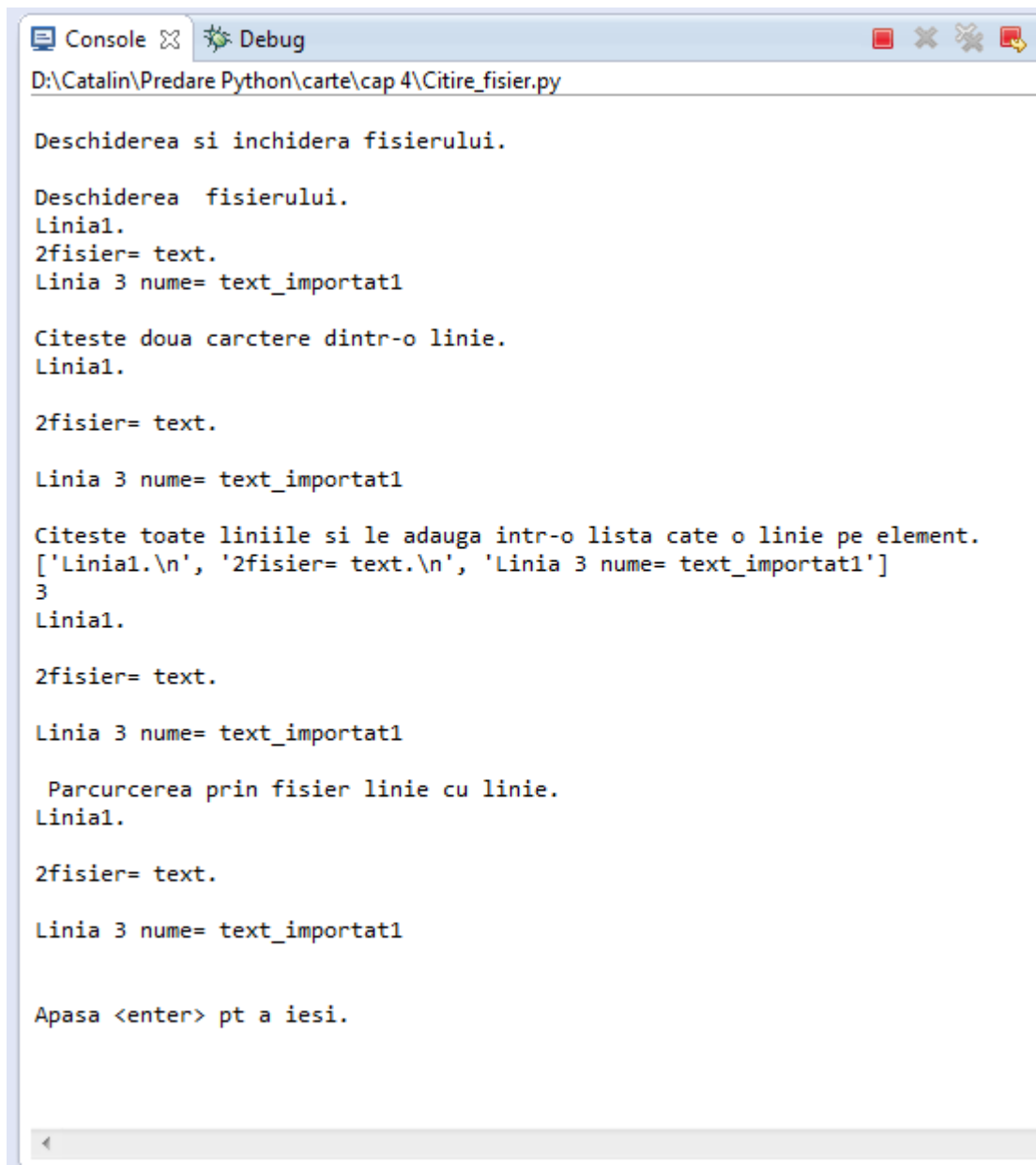
print "\nDeschiderea și inchiderea fisierului."
text_file = open("text_importat1.txt", "r")
text_file.close()

print "\nDeschiderea fisierului."
text_file = open("text_importat1.txt", "r")
print text_file.read() #citeste toate liniile intr-un sir de caracter
text_file.close()

print "\nCiteste doua carctere dintr-o Linie."
text_file = open("text_importat1.txt", "r")
print text_file.readline() # Prima linie
print text_file.readline() # A doua linie
print text_file.readline() # A treia linie
text_file.close()

print "\nCiteste toate liniile și le adauga intr-o lista cate o linie pe element."
text_file = open("text_importat1.txt", "r")
linii = text_file.readlines()
print linii
print len(linii)
for linie in linii:
    print linie
text_file.close()

raw_input("\n\nApasa <enter> pt a iesi.")
```



The screenshot shows a Python IDE window with a 'Console' tab. The file path is 'D:\Catalin\Predare Python\carte\cap 4\Citire_fisier.py'. The console output shows the execution of a program that reads a file line by line. The output is as follows:

```
Deschiderea si inchiderea fisierului.  
  
Deschiderea fisierului.  
Linia1.  
2fisier= text.  
Linia 3 nume= text_importat1  
  
Citeste doua caractere dintr-o linie.  
Linia1.  
  
2fisier= text.  
  
Linia 3 nume= text_importat1  
  
Citeste toate liniile si le adauga intr-o lista cate o linie pe element.  
['Linia1.\n', '2fisier= text.\n', 'Linia 3 nume= text_importat1']  
3  
Linia1.  
  
2fisier= text.  
  
Linia 3 nume= text_importat1  
  
Parcurcerea prin fisier linie cu linie.  
Linia1.  
  
2fisier= text.  
  
Linia 3 nume= text_importat1  
  
Apasa <enter> pt a iesi.
```

Fig. 27

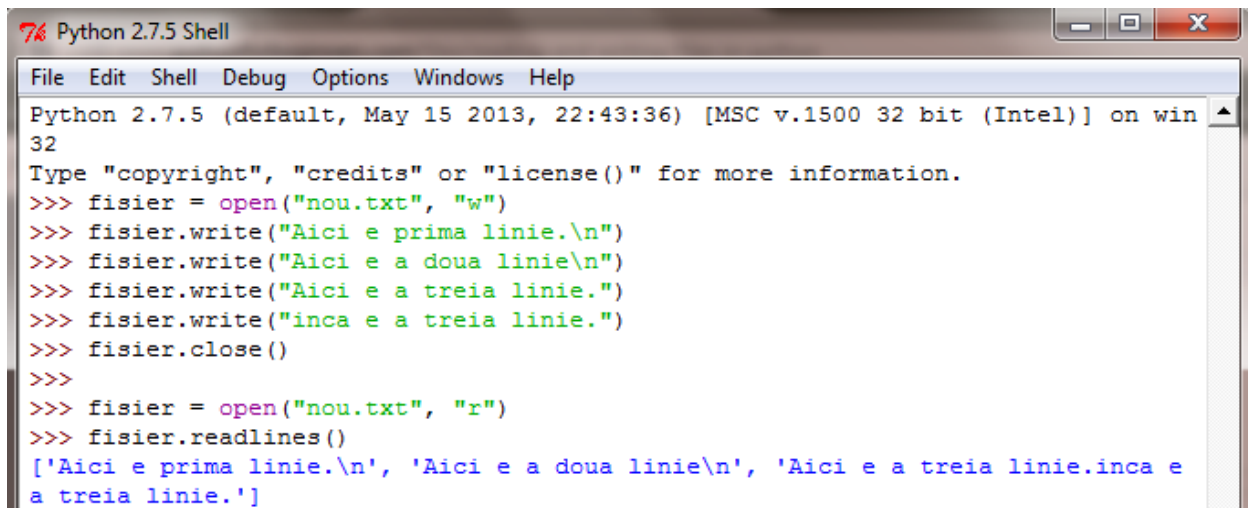
Scrierea fișierului se realizează cu `write()`.

Iată un mic exemplu mai jos:

```
fișier = open("nou.txt", "w")  
fișier.write("Aici e prima linie.\n")  
fișier.write("Aici e a doua linie\n")
```



```
fisier.write("Aici e a treia linie.")
fisier.write("inca e a treia linie.")
fisier.close()
```

A screenshot of a Python 2.7.5 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The title bar says 'Python 2.7.5 Shell'. The main text area shows the following code and its output:

```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> fisier = open("nou.txt", "w")
>>> fisier.write("Aici e prima linie.\n")
>>> fisier.write("Aici e a doua linie\n")
>>> fisier.write("Aici e a treia linie.")
>>> fisier.write("inca e a treia linie.")
>>> fisier.close()
>>>
>>> fisier = open("nou.txt", "r")
>>> fisier.readlines()
['Aici e prima linie.\n', 'Aici e a doua linie\n', 'Aici e a treia linie.inca e
a treia linie.']
```

Fig. 28

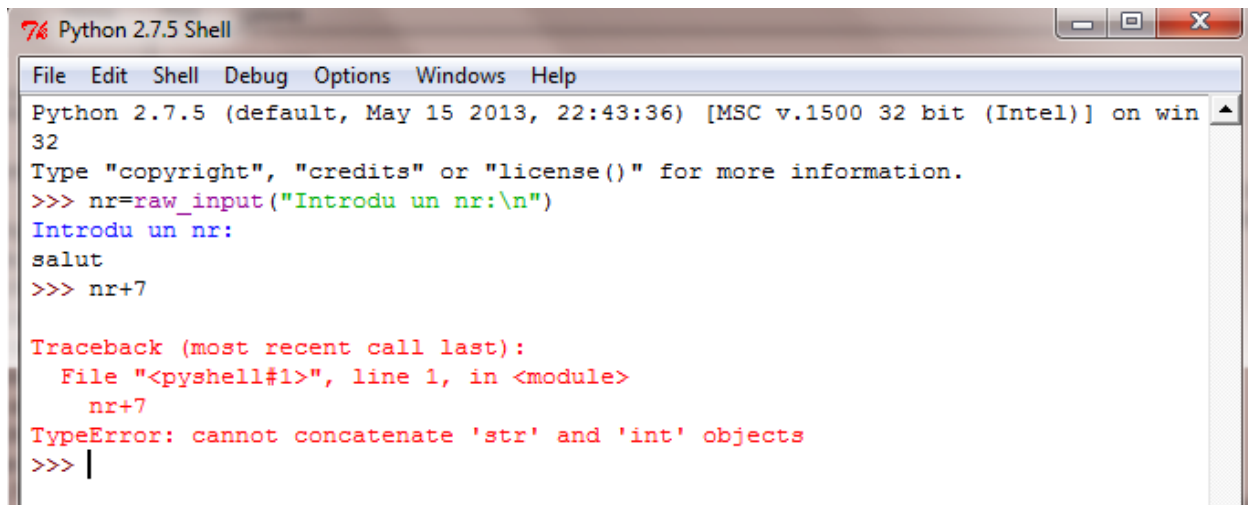
In Fig. 28 putem vedea ca adaugarea in sirul de caractere a secvente de evadare `<\n>` este obligatorie pentru ca sirul de caractere sa treaca pe linia urmatoare.

Adaugarea de caractere este controlata si de modul ales la crearea obiectului „open file”. Daca sub clasa open alegem „w” atunci vom rescrie fisierul anterior, daca alegem „a”, vom adauga la final aceste caractere.

Excepții

Când Python întâlnește o eroare, interpretorul va ridica acea eroare și o va afișa. Python are un sistem de eroare foarte bine pus la punct, spre deosebire de alte limbaje de programare cu care este des comparat cum ar fi spre exemplu de TCL ce ascunde erorile încapsulate. Când fac referire la erorile încapsulate mă refer la o eroare care apare într-o clasa ce a fost apelata de alta clasa etc. Python arata fiecare apelare în parte pana la eroarea apăruta.

O excepție este o suprimare a unei erori. Suprimarea erorilor este necesara cu precădere în cazurile când userul introduce de la tastatura anumite elemente.

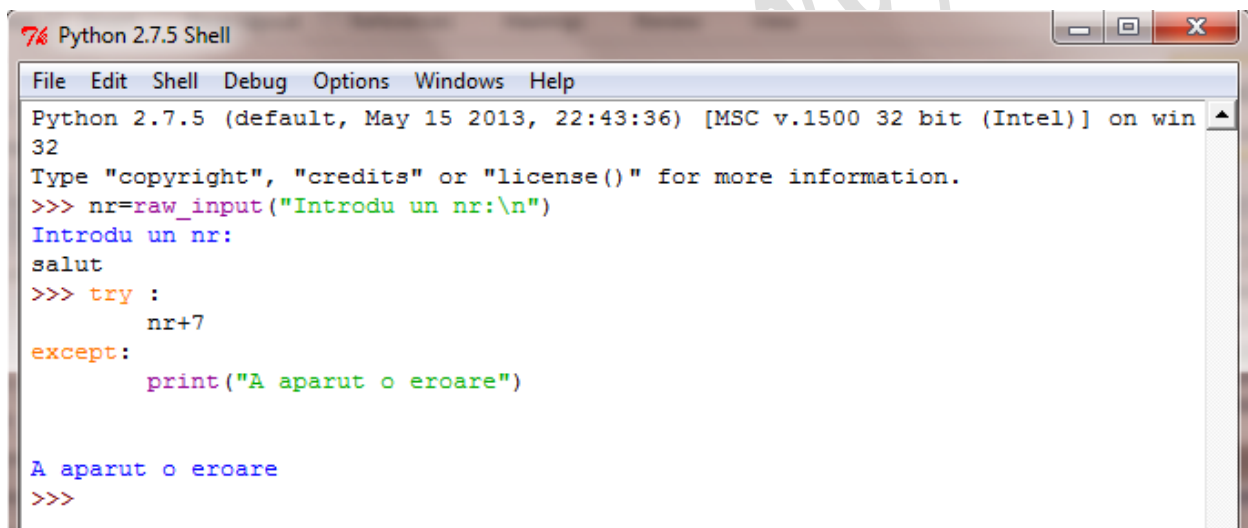


```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> nr=raw_input("Introdu un nr:\n")
Introdu un nr:
salut
>>> nr+7

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    nr+7
TypeError: cannot concatenate 'str' and 'int' objects
>>> |
```

Fig. 29

Pentru a suprima o eroare putem folosi try except. Blocul de sintaxe except va rula doar daca exista o eroare in rularea bolcului de sintaxe de sub try.

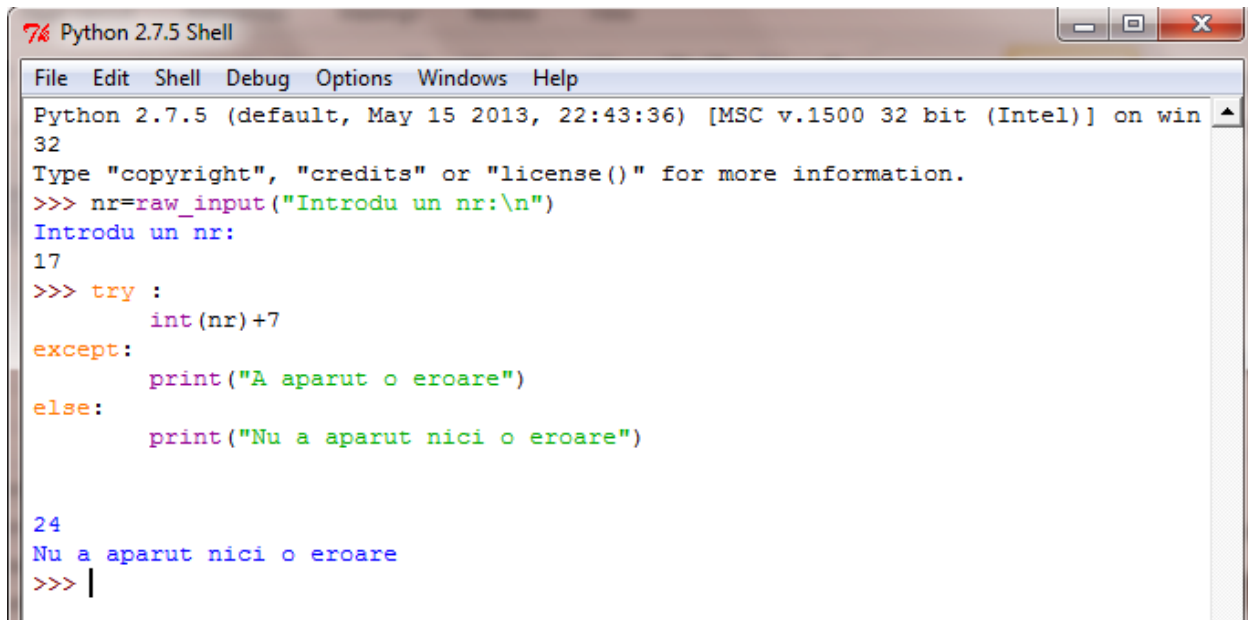


```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> nr=raw_input("Introdu un nr:\n")
Introdu un nr:
salut
>>> try :
    nr+7
except:
    print("A aparut o eroare")

A aparut o eroare
>>>
```

Fig. 30

O alta utilizare este try/except/else. Blocul de sintaxe else va rula doar dacă nu se va ridica nici o eroare si nu va fi rulat daca apare o eroare in blocul try.



```

Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> nr=raw_input("Introdu un nr:\n")
Introdu un nr:
17
>>> try :
        int(nr)+7
except:
    print("A aparut o eroare")
else:
    print("Nu a aparut nici o eroare")

24
Nu a aparut nici o eroare
>>> |

```

Fig. 31

În cazul în care scrie except, orice eroare va fi generată și va fi suprimată. Totuși poate ne dorim ca doar erorile de un anumit tip să fie suprimate. Spre exemplu un text introdus de la tastatură este de tipul șir de caractere. Dacă încercăm să concatenăm un șir de caractere cu un număr ne va rezulta `TypeError` (vezi Fig.29) sau dacă încercăm să convertim acel șir de caractere într-un număr ne va da `ValueError` (vezi Fig. 32). Prin urmare, există multe erori pentru fiecare caz. Mai jos regăsim un tabel cu fiecare eroare în parte.

```

>>> int("salut")

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    int("salut")
ValueError: invalid literal for int() with base 10: 'salut'
>>> |

```

Fig. 32

Tipul Excepției	Descriere
IOError	Eroare ridicată când încercăm să deschidem un fișier inexistent.

Tipul Excepției	Descriere
IndexError	Eroare ridicata când o secvența este indexata cu un număr inexistent.
KeyError	Eroare ridicata când o cheie a unui dictionar nu este găsită.
NameError	Eroare ridicata când un nume de variabila sau funcție nu este găsit.
SyntaxError	Eroare ridicata când o eroare de sintaxa apare.
TypeError	Eroare ridicata când o funcție incorporata este aplicata la un obiect/variabila neadecvata.
ValueError	Eroare ridicata când o funcție incorporata este aplicata la un obiect/variabila adecvata, dar o valoare greșită.
ZeroDivisionError	Eroare ridicata când al doilea atribut al unei diviziuni(numitor) este zero.

Mai jos se poate observa exemplul de mai sus aplicat doar la ValueError. Putem aplica except doar pentru o eroare data între paranteze, nu pentru toate erorile. Adicional la acest exemplu arătăm și o tehnică de capturare a erorilor. Erorile se capturează cu ajutorul unei variabile adăugate după parantezele sintaxei except, iar virgula nu trebuie să lipsească. În Fig.33 aceasta variabilă este <<e>> și va stoca corpul erorii aparute.

```

>>> try:
>>>     int("salut")
>>> except (ValueError), e:
>>>     print "nu da eroare"

nu da eroare
>>> print e
invalid literal for int() with base 10: 'salut'
>>> # variabila e a capturat mesajul de eroare
>>> |

```

Fig. 33

În cazul în care doriți să luați câte o decizie pentru fiecare eroare în parte o expresie try poate avea mai multe except-uri ce se aplică la o eroare diferită. În Fig.34 putem vedea că variabila x nu există. La apelarea variabilei x ne returnează `TypeError: name x is not defined`. Dacă implementez o construcție try-except ce va conține două except-uri vom vedea că putem aplica diferite blocuri de sintaxă pentru fiecare eroare apărută. Astfel va rula doar blocul de sintaxă de sub `except (NameError)` deoarece eroarea generată în try este `NameError`.

```
>>> x

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> try:
    x
except (TypeError):
    print "TypeError"
except (NameError):
    print "NameError"

NameError
>>>
```

Fig.34

Atenție totuși la eroarea apărută. Dacă eroarea nu există într-o paranteză a unui `except` atunci nu vom excepta apariția acelei erori. În Fig.35 putem vedea că variabila x nu există. La apelarea variabilei x ne returnează `TypeError: name x is not defined`. Dacă implementez o construcție try-except ce va conține două except-uri, dar nici unul din acele except-uri nu conține eroarea întâlnită în try (aici `NameError`), nu vom face excepție de la eroarea respectivă.

```

>>> x

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> try:
    x
except (TypeError):
    print "TypeError"
except (IndexError):
    print "IndexError"

Traceback (most recent call last):
  File "<pyshell#16>", line 2, in <module>
    x
NameError: name 'x' is not defined
>>>

```

Fig.35

Un ultim exemplu in ceea ce priveste exceptiile se regaseste mai jos. In fig. 36 cream u variabila x ce stocheaza o lista cu doua elemente. Variabila y nu exista. Cream apoi o constructie try-else care va genera eroarea NameError deoarece in try se regaseste variabila y. Intre parantezele primului except nu regasim NameError deci nu vom rula print „IndexError,KeyError”. Al doilea except se aplica la toate erorile si il vom folosi ca o plasa de siguranta. Al doilea except va rula pentru toate erorile aparute sub try ce nu sunt IndexError sau KeyError.

```

>>> x=[1,2]
>>> y

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> try:
    y
except (IndexError,KeyError):
    print "IndexError,KeyError"
except:
    print "altceva"

altceva

```

Fig.36

Fig. 37 ne prezinta o continuare a codului prezentat in Fig.36. Astfel daca am apela x[3] ne va ridica eroarea IndexError deoarece x este o lista cu doua elemente, deci cu un index doar de 0 sau 1. Vedem ca in acest caz eroarea IndexError se regaseste in primul except, deci va rula print „IndexError,KeyError”.

```
>>> x[3]

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    x[3]
IndexError: list index out of range
>>> try:
        x[3]
except (IndexError,KeyError):
    print "IndexError,KeyError"
except:
    print "altceva"

IndexError,KeyError
>>>
```

Fig.37