Programarea calculatoarelor si limbaje de programare II

Clase: notiuni de baza

Universitatea Politehnica din Bucureşti

Sumar

- ***Programarea orientata obiect, OOP***
- Programarea cu clase in Python
- Exemplu pentru seminar

Introducere

- Pana acum am facut o programare cu obiecte predefinite – *int, list, str, tuple, set, dict*
  - le-am folosit in expresii, am apelat metode, etc
- OOP (**O**bject – **O**riented **P**rogramming), programarea orientata obiect, presupune folosirea si a ideii de mostenire ierarhica – *inheritance hierarchy*
- In Python, obiectul de tip **class** implementeaza OOP: factorizarea codului, reducerea redundantei, adaptarea codului existent in loc de modificare in-place.
- Clasele sunt pachete de functii care folosesc si prelucreaza obiecte predefinite
  - in plus, ele creeaza obiecte noi bazate pe mecanismul de mostenire

Rolul claselor

Clasele folosesc elemente ale OOP:

- **Mostenirea** – clasele mostenesc proprietati generale ale superclaselor, implementate o singura data, si refolosite de subclase
- **Compozitia** – clasele contin si alte obiecte, eventual clase, care le determina comportamentul si relatiile cu alte obiecte
  - Exemplu: in sistemele GUI, interfetele se proiecteaza cu widgets: butoane, etichete, etc, care sunt desenate pe ecran odata cu containerul lor (*compozitie*). Particularizarea widget-urilor, cu fonte, text, culori speciale reprezinta adaptarea diverselor interfete (*mostenire*)

Clasele sunt unitati de program – ca functiile si modulele, ce definesc spatii de nume:

Rolul...

- **Instante multiple** – apelul de clase creeaza instante noi, cu spatiu de nume propriu si atribute preluate de la clasa; analog cu fabricile de functii, clasele ofera memorarea starii per apel – instanta
- **Particularizare via mostenire** – subclasele mostenesc de la superclase atribute pe care le pot redefini, rezultand o ierarhie de spatii de nume cu atribute disponibile pentru instantele unei clase
- **Inlocuirea operatorilor** (*Operator overloading* sau *overriding*) – clasele pot efectua operatii precum cele suportate de obiecte predefinite: decupare(*slicing*), concatenare, indexare, afisare, etc

Rolul...

Clasele folosesc doua mecanisme:

1. metode cu un prim argument, numit *self* – prin convention, ce refera instanta curenta
2. cautarea atributelor mostenite

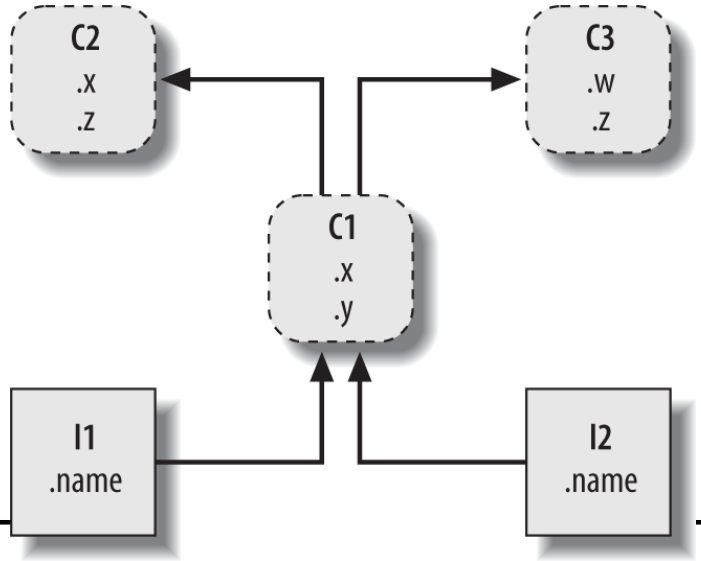
Cautarea atributelor mostenite

Se bazeaza pe expresia:

obiect.atribut

In cazul unui *obiect* derivat dintr-o clasa, cautarea atributului se face intr-un arbore de obiecte inlantuite, pana la prima aparitie a atributului

- **atributul este cautat in obiect, apoi in clasele de deasupra, de jos in sus si de la stanga la dreapta:**



C1, C2, C3 – clase

I1, I2 – instante (ale clasei C1)

Cautarea...

Clasele:

- sunt fabrici de instante carora le furnizeaza atribute comportamentale, e.g. functii – atribut de tip metoda

Instantele:

- reprezinta obiecte dintr-un program, cu valori ale atributelor specifice fiecaruia

- O instanta mosteneste atributele sale de la clasa din care deriva
- O clasa mosteneste atribute de la clasele de deasupra sa din arbore
- Clasele mai sus in arbore sunt superclase – C2 si C3
- Clasele de mai jos sunt subclase – C1

Cautarea...

- Superclasele ofera comportament partajat de toate subclasele
  - Deoarece cautarea se face de jos in sus, subclasele pot inlocui (override) atributele din superclase prin redefinire mai jos in arbore
- Expresia **I2.w** cauta atributul **w** in ordine in: **I2, C1, C2, C3**
  - Cautarea se opreste la prima aparitie – sau se raporteaza o eroare daca nu este gasit
- Atributele mostenite de cele doua instante sunt: **w**, **x**, **y** si **z**

Cautarea...

- Alte atribute:
  - **I1.x** si **I2.x** sunt gasite in **C1** – mai jos decat C2
  - **I1.y** si **I2.y** sunt gasite in **C1** – singurul loc unde apare **y**
  - **I1.z** si **I2.z** sunt gasite in **C2** – aflata mai la stanga decat C3
  - **I1.name** si **I2.name** sunt gasite imediat in **I1**, respectiv **I2**

Clase si instante

- Clasele si instantele sunt obiecte cu spatiu de nume, prevazut cu atribute
  - Seamana cu modulele, insa:
    - au acces si la alte spatii de nume din arborele de clase
    - corespund unei instructiuni de tip *class*, nu intregului fisier
- Clasele sunt fabrici care produc instante
  - Modulele au o singura instanta/reprezentare in memorie – ce se poate modifica cu *reload()*
    - instantele unei clase pot fi oricat de multe
- Clasele au functii asociate (numite metode)
  - Instantele au de obicei date, folosite de functii
    - modelul OOP: instantele sunt ca inregistrari de date, iar clasele sunt ca programe de prelucrare; in plus exista mostenirile ierarhice care permit personalizarea codului

Apelul de metode

- Metodele sunt functii atasate claselor ca atribute ale acestora
- Apelul unei metode, e.g. *I2.w()*, presupune intotdeauna aplicarea acesteia asupra unei instante a clasei
  - **I2.w()** este convertita/echivalenta la/cu **C3.w(I2)**
- Primul argument al metodei este conventional numit **self** si este o referinta catre instanta clasei, asupra careia se executa functia
  - Apelul **I2.w()** presupune:
    1. cautarea metodei **w()** pe arborele de mostenire
    2. asocierea instantei **I2** cu apelul functiei **w()** prin primul sau argument, *self*
  - Apelul **C3.w(I2)** realizeaza cei doi pasi de mai sus manual

Arbori de clase

- Se construiesc cu instructiunea **class**:
  - Fiecare instructiune *class* genereaza un nou obiect de tip clasa
  - Fiecare apel de clasa genereaza un nou obiect de tip instanta
  - Instantele sunt asociate cu clasa din care au fost create
  - Clasele sunt asociate cu superclasele precizate intre paranteze rotunde in antetul instructiunii *class*:
    - ordinea din lista superclaselor este ordinea de la stanga la dreapta in arborele de mostenire
- Clasele se plaseaza in module, la fel ca functiile, si instructiunea *class* se executa cu ocazia importarii modulelor, e.g.:

```
class C2: ... # Creare obiect de tip clasa
class C3: ...
class C1(C2, C3): ... # C1 deriva din C2 si C3, in
    aceasta ordine
```

```
I1 = C1() # Crearea instantei, I1 si apoi I2
I2 = C1() # asociate
    cu clasa C1
```

Arbori...

- In cazul lui *C1* mostenirea este multipla:
  - Daca sunt listate mai multe clase intre parantezele rotunde, ordinea lor, de la stanga la dreapta, este ordinea cautarii atributelor mostenite de la aceste superclase
    - Implicit este numele cel mai din stanga, insa poate fi oricand ales din clasa in care exista, e.g. **C3.z**
- Atributele:
  - Sunt atasate claselor prin asignari facute in corpul clasei, din afara *def*-urilor din clase; sunt partajate de subclase si instante.
  - Sunt atasate instantei prin asignare – calificare cu primul argument, *self*, al functiilor din interiorul claselor.

Arbori...

- Exemplu:

```
class C2: ... # Superclasa C2
class C3: ... # Supeclasa C3
class C1(C2, C3): # Subclasa C1
    def setname(self, who): # Asignare: C1.setname
        self.name = who # self e fie I1 fie I2

I1 = C1() # Crearea instantei I1
I2 = C1() # Crearea instantei I2
I1.setname('bob') # I1.name = 'bob'
I2.setname('sue') # I2.name = 'sue'
print(I1.name) # Afiseaza 'bob'
```

- ***def***-ul din interiorul clasei se numeste metoda:
  - are automat un prim argument – numit *self* prin conventie – care este referinta catre instanta de procesat
  - argumentele actuale (de la apelul metodei) urmeaza dupa *self* (who)
- Atributele instantei si cele ale clasei nu sunt declarate ci create/modificate prin asignare

Arbori...

- Atributele lui *self* sunt create/modificate prin asignare si sunt cele ale instantei celei mai de jos din arbore
- Orice atribut din arbore poate fi accesat/setat daca este vizibil din spatiul de nume curent, e.g. *C1.setname(self,...)* sau *I1.setname(...)*

Inlocuirea operatorilor

- Metodele care inlocuiesc operatori:
  - Sunt mostenite ca orice metoda
  - Sunt optionale:
    - daca lipsesc, operatia nu este suportata
  - Au nume care incepe si se termina cu dublu underscore
  - Sunt executate automat de Python cand instantele care le au ca atribute apar in operatii corespunzatoare
  - Clasele care au astfel de metode seamana cu tipurile de date predefinite
  - Exemple de (*operator overloading*) metode:
    - **__and__** este folosita pentru intersectia de multimi cu **&**
    - **__add__** este pentru adunare/concatenare cu **+**
- Constructorul clasei este metoda **__init__**

Inlocuirea...

- **__init__** este apelat cand o instanta este creata, fiind folosit la initializarea (tuturor) atributelor unei instante
  - argumentul *self* este instanta creata
  - argumentele de la apelul clasei, intre parantezele rotunde, vor fi celelalte argumente actuale ale lui **__init__**

```
class C2: ... # Superclase
class C3: ...
class C1(C2, C3):
    def __init__(self, who): # Constructor
        self.name = who # self e fie I1 fie I2

I1 = C1('bob') # I1.name = 'bob'
I2 = C1('sue') # I2.name = 'sue'
print(I1.name) # Afiseaza 'bob'
```

- Fata de metoda *setname,* apelul lui *__init__* este garantat, odata cu crearea instantei

Reutilizarea codului cu OOP

- OOP permite:
  - mostenirea
  - gama larga de operatori inlocuiti:
    - indexare, selectia atributelor, afisare (print), etc.
  - In esenta este vorba despre cautarea atributelor in arbori si functii cu un prim argument special
  - reutilizarea codului, in loc de editare/modificari in-place sau rescriere de la inceput
    - se efectueaza prin redefinirea metodelor intr-o subclasa si refolosirea metodelor superclasei
  - Clasele sunt mai capabile decat modulele sau functiile
    - sunt structuri care impacheteaza logica si datele, usurand depanarea programelor

Polimorfism cu clase

- Fie o superclasa care implementeaza angajatii unei organizatii:

```
class Employee: # Superclasa
    def computeSalary(self): ... # Valabila pentru orice angajat
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

- Fie o subclasa pentru anumite grupuri de angajati, care personalizeaza anumite metode si mosteneste celelalte de la superclasa:

```
class Engineer(Employee): # Subclasa particulara
    def computeSalary(self): ... # Ceva diferit de restul angajatilor
bob = Employee() # Angajat
sue = Employee() # Angajat
tom = Engineer() # Salariu calculat in mod diferit
```

  - Metoda *Engineer.computeSalary* este mai jos in arbore, deci va inlocui pe cea din *Employee*

Polimorfism...

- Cele trei instante pot fi obiecte dintr-o lista cu angajati:

```
company = [bob, sue, tom] # O lista de obiecte
for emp in company:
    print(emp.computeSalary()) # Versiunea rulata depinde de clasa obiectului
```

- Acest comportament – semnificatia unei operatii depinde de obiectele asupra carora se exercita – se numeste **polimorfism**
- Polimorfismul permite incapsularea/ascunderea diferentele de implementare ale unor interfete comune:

```
def processor(reader, converter, writer): #
    Procesare de obiecte cu interfata formata
    din metodele read() si write()
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Polimorfism...

- Functia *processor*() suporta subclase care specializeaza operatia de citire, *read* (sau si *write*):

```
class Reader:
    def read(self): ... # Comportament implicit
    def other(self): ...
class FileReader(Reader):
    def read(self): ... # Citire de fisier
class SocketReader(Reader):
    def read(self): ... # Citire din retea
```

```
...
processor(FileReader(...), Converter,
    FileWriter(...))
processor(SocketReader(...), Converter,
    TapeWriter(...))
processor(FtpReader(...), Converter,
    XmlWriter(...))
```

Programare prin specializarea codului

- In diverse domenii exista deja colectii de superclase ce alcatuiesc un *framework*, gata de a fi intrebuintate in aplicatii proprii, e.g. interfete pentru:
  - baze de date
  - testarea programelor
  - GUI *toolkit*
- Clasele dintr-un *framework* implementeaza deja majoritatea codului necesar, ramanand doar scrierea a catorva metode in cateva subclase
- Sabloane de proiectare – *design patterns* – sunt utile in abordarea OOP

Sumar

- Programarea orientata obiect, OOP
- ***Programarea cu clase in Python***
- Exemplu pentru seminar

Generarea de instante multiple

- In OOP cu Python exista doua categorii de obiecte:
  1. Clase
     - care produc obiecte de tip instanta, furnizandu-le comportamentul implicit
     - sunt definite cu instructiunea **class**
  2. Instante
     - sunt spatii de nume reale cu acces la atributele clasei din care provin
     - sunt create cu apeluri de clase
- Clasele sunt fabrici de instante multiple.
- Exista insa, doar o copie a fiecarui modul importat – actualizata cu functia *reload()*
- Fiecare instanta are date proprii/diferite, reprezentand versiuni multiple ale obiectului clasa modelat

Generarea...

- Instantele se aseamana cu functiile fabricate, starea fiind memorata cu atribute in mod explicit (in loc de implicit, cu referinte in spatiul de nume al functiei producatoare)
- Clasele sunt mai complexe – suporta specializarea prin mostenire, inlocuirea operatorilor, comportament variabil prin metode

Obiecte de tip clasa cu comportament implicit

- Instructiunea **class** creeaza un obiect de tip clasa si ii asigneaza un nume
  - este o instructiune executabila (ca *def*)
  - numele clasei se afla in antetul instructiunii *class*
  - instructiunea *class* se executa (de obicei) la importarea modulului ce o cuprinde
- Atribuirile din corpul clasei (din afara oricarui *def*) produc atributele clasei
  - care sunt accesibile prin calificare: **object.name** (dupa crearea obiectului de tip clasa)
- Atributele clasei reprezinta starea obiectului si comportamentul sau
  - instantele partajeaza atributele clasei, inclusiv metodele (instructiuni *def* din corpul clasei)

Obiecte de tip instanta a unei clase

- Apelarea unei clase – ca o functie – creeaza o instanta noua a clasei (obiect de tip instanta)
- Fiecare instanta mosteneste atributele clasei si are si propriul spatiu de nume, initial vid apoi populat cu referinte catre atributele clasei
- Asignarile catre atributele lui *self* in metode creeaza atribute per fiecare instanta – **nu** in clasa
  - ***self*** este primul argument al metodelor si refera instanta procesata de metoda
- Astfel clasele definesc date si metode partajate de instantele generate – care:
  - sunt obiecte concrete in aplicatii ce pot inregistra date per instanta, ce pot fi diferite

O prima clasa

- Fie o clasa numita *FirstClass*:

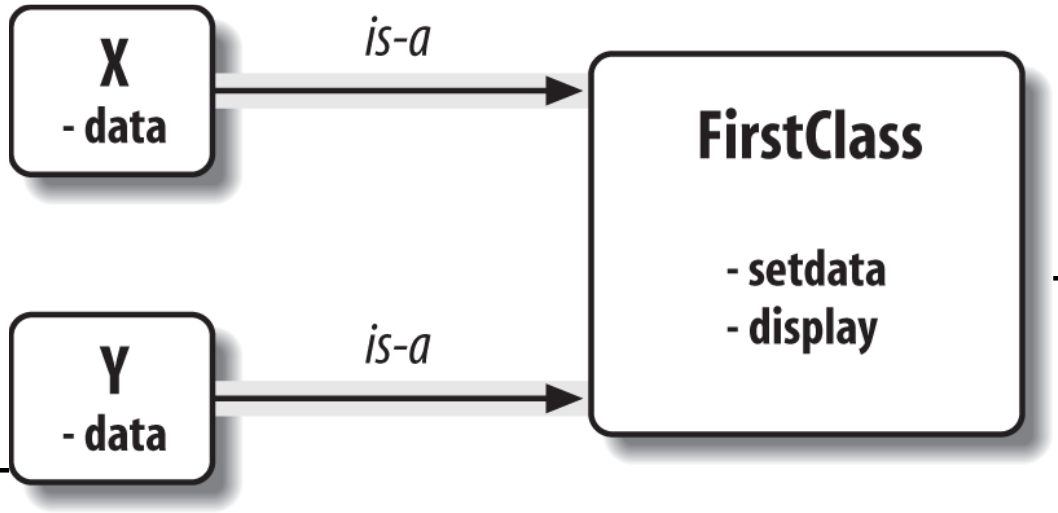
```
>>> class FirstClass:                    # Antetul clasei
        def setdata(self, value):         # Metoda a clasei
            self.data = value             # self este instanta curenta
        def display(self):                # Alta metoda
            print(self.data)              # self.data: per instanta
```

- Instructiunea *def* este o asignare:
  - *setdata* si *display* devin atribute ale clasei: *FirstClass.setdata* si *FirstClass.display*
- Orice asignare in/la nivelul clasei devine atribut al clasei

O...

- Metodele – sunt functii din corpul clasei, suporta valori implicite, returneaza valori sau genereaza rezultate la cerere cu *yield*, etc.
  - Primul argument al metodei primeste in mod automat o referinta catre instanta, la apel

```
>>> x = FirstClass() # Instanta noua                    # Apelul clasei genereaza instante noi
>>> y = FirstClass() # Alta instanta, spatiu de nume nou   # Au fost create trei obiecte, o clasa si doua instante ale acesteia:
```



O...

- Cele doua instante nu contin nimic initial, dar selectiile din instanta cu nume din clasa sunt gasite in clasa (cu exceptia aflarii lor si in instanta):

```
>>> x.setdata("King Arthur") # Apelul metodei    >>> y.setdata(3.14159) # Apelul metodei
    FirstClass.setdata( x, "King Arthur" ), self e x      FirstClass.setdata(y, 3.14159), self este y
```

- Metodele *setdata* si *display* sunt gasite prin mostenire din clasa
  - mostenirea se produce in momentul calificarii atributelor si se bazeaza pe cautarea in obiectele inlantuite (in arbore)
- Asignarea lui *self.data* se face in spatiul de nume al instantei (**nu** al clasei)
- Distinctia intre mai multe instante se face cu argumentul *self*:

O...

```
>>> x.display() # self.data difera intre instante
King Arthur
```

```
>>> y.display() # Se executa apel de
        FirstClass.display(y)
3.14159
```

- Daca *setdata* nu este apelata inainte de *display* atributul *data* nu este inca creat prin asignare
- Modificarea atributelor de instanta se poate face:
  - din interiorul clasei prin calificarea lui *self* in metode
  - din exteriorul clasei, prin calificarea directa a obiectului instanta:

```
>>> x.data = "New value" # Citire/scriere
    atribute de instanta, in afara codului clasei
```

```
>>> x.display()
New value
```

- Se pot chiar adauga atribute noi instantei:

```
>>> x.anothername = "spam"     # Setare de atribut nou!
```

Specializarea mostenirii claselor

- Clasele:
  - Pot fi fabrici de instante
  - Permit si crearea de subclase – in loc de a se modifica componente in-place
- Clasele pot mosteni de la alte clase (precum instantele mostenesc de la clase)
  - Se creeaza ierarhii de clase, cu comportament **specializat** prin redefinirea atributelor in subclase, deci mai jos, inlocuindu-se cele din superclase (de mai sus)
  - Modulele au un spatiu de nume unic si plat...
- Reguli de mostenire:
  - Clasele mostenesc atribute de la superclase
    - daca atributele lipsesc, ele pot fi gasite atunci cand sunt accesate

Specializarea...

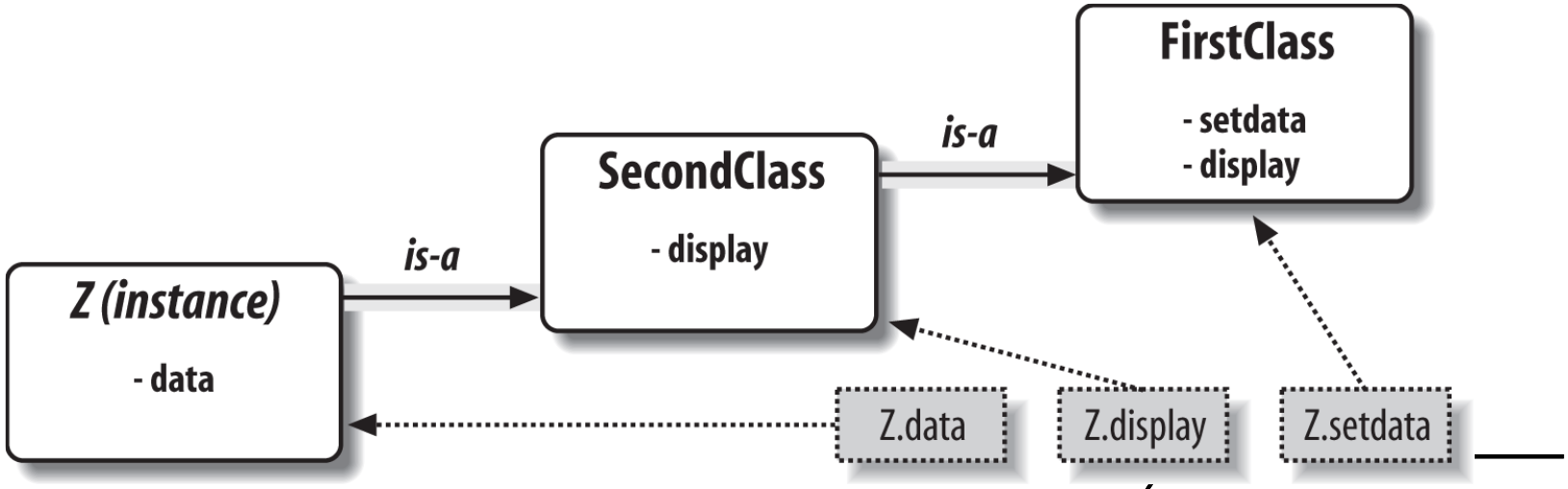
- Instantele mostenesc atribute de la toate clasele accesibile
  - atributele sunt cautate in instanta, apoi clasa, apoi superclase (de jos in sus si de la stanga la dreapta)
- Fiecare calificare **obiect.atribut** produce o cautare noua, independenta
  - la fel in afara clasei, e.g. **X.atribut** si in interior, e.g. **self.atribut**
- Modificarea codului se face prin subclasare, nu prin modificarea superclaselor
  - numele redefinite in subclase le inlocuiesc si specializeaza pe cele din superclase

- Astfel, clasele realizeaza:
  - factorizarea si specializarea codului
  - minimizarea redundantelor
  - specializarea codului existent in loc de rescriere sau modificare in-place

A doua clasa

- Fie o clasa numita *SecondClass* care mosteneste toate atributele de la *FirstClass* si redefineste metoda *display*:

```
>>> class SecondClass(FirstClass):        # Mosteneste setdata
        def display(self):                # Redefineste/inlocuieste/specializeaza display
            print('Current value = "%s"' % self.data)
```



A...

- Inlocuirea atributelor prin redefinire mai jos in arbore se mai numeste *overloading*:

```
>>> z = SecondClass()                          >>> FirstClass.display(z) # Vechiul display
>>> z.setdata(42) # setdata din FirstClass     42
>>> z.display() # Metoda display din SecondClass>>> x.display() # x este tot instanta lui FirstClass
Current value = "42"                           New value
```

Clasele sunt atribute ale modulelor

- Clasa *FirstClass* codificata intr-un modul:

```
from modulename import FirstClass          # Importarea lui FirstClass din modulename
class SecondClass(FirstClass):             # Folosirea numelui in mod direct
    def display(self): ...
```

```
import modulename                          # Importarea intregului modul  modulename
class SecondClass(modulename.FirstClass):  # Cu calificare
    def display(self): ...                 # Cod echivalent celui de mai sus
```

- Modulele pot include diverse clase, functii, variabile, care devin atribute de modul (cand se executa codul modulului – prin importare sau ca script principal):

```
# Fisierul food.py:
var = 1 # food.var
def func(): ... # food.func
class spam: ... # food.spam
class ham: ... # food.ham
class eggs: ... # food.eggs
```

Clasele...

- Clasa poate avea acelasi nume ca modulul din care face parte:

*# Fisierul **person.py**:*	*# Folosirea clasei din modulul person*
class person: ...	**import person**
	x = person.person()
from person import person	*# Mai clar*
x = person()	*# Numele de clasa*

- Se recomanda ca numele de clasa sa inceapa cu o litera mare, iar numele de modul – cu litere mici:

import person	*# Litere mici pentru module*
x = person.Person()	*# Capitalizarea numelui de clasa*

- Modulele sunt cuprinse intr-un fisier, iar *class* este doar o instructiune din fisier

Interceptarea operatorilor in Python

- Obiectele de tip clasa pot intercepta si efectua operatii specifice tipurilor de date predefinite:
  - adunare, decupare, afisare, calificare, etc
- Inlocuirea (overloading) operatorilor:
  - Metode cu dublu underscore, __**X**__, avand nume speciale, prestabilite, se folosesc la interceptarea operatiilor
  - Aceste metode sunt apelate automat cand instantele apar in operatii predefinite, e.g. __**add**__ pentru o expresie cu **+**
  - Majoritatea operatorilor predefiniti pot fi inlocuti
  - Absenta metodelor care inlocuiesc operatori inseamna ca operatia nu este suportata (nu exista metode implicite)
  - Clasele din v3.X ofera cateva metode predefinite, de tipul __**X**__, mostenite de la superclasa ***object***

Interceptarea...

- Operatorii integreaza clasele in modelul obiectelor din Python
  - operatorii inlocuiti in clase permit claselor sa se comporte ca obiectele predefinite, cu interfete compatibile
- Inlocuirea operatorilor este optionala
  - Metoda *giveRaise* are mai multa semnificatie ca opertorul **+**
- Metoda **__init__**, asa-zisul constructor al clasei, este prezenta in majoritatea claselor, avand rolul de a crea atributele fiecarei instante, impreuna cu argumentul *self*

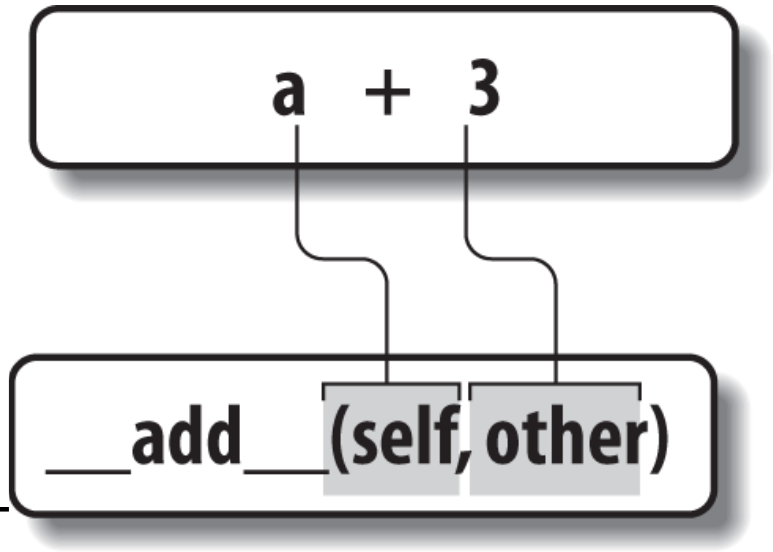
A treia clasa

- Clasa *ThirdClass* este o subclasa a lui *SecondClass*:
  - **__init__** este executat la crearea instantei – *self*
  - **__add__** se executa cand o instanta a clasei ThirdClass apare intr-o expresie cu operatorul **+**
  - **__str__** se executa la afisarea obiectelor – cand obiectul este convertit la un *str* afisabil de functia *print*, sau cu functia predefinita *str()*

```
>>> class ThirdClass(SecondClass): # Mosteneste pe SecondClass
        def __init__(self, value): # Apelata de "ThirdClass(value)"
            self.data = value
        def __add__(self, other): # La "self + other"
            return ThirdClass(self.data + other)
        def __str__(self): # La "print(self)", " sau la str()"
            return '[ThirdClass: %s]' % self.data
        def mul(self, other): # Metoda cu nume, in-place, nu operator inlocuit
            self.data *= other
```

A...

```
>>> a = ThirdClass('abc') # Apel de ThirdClass.__init__(self, 'abc')
>>> a.display() # Metoda mostenita
Current value = "abc"
>>> print(a) # __str__: stringul de afisare
[ThirdClass: abc]
>>> b = a + 'xyz' # __add__: returneaza o instanta noua a lui ThirdClass
>>> b.display() # b are toate metodele lui ThirdClass
Current value = "abcxyz"
>>> print(b) # __str__: stringul de afisare
[ThirdClass: abcxyz]
>>> a.mul(3) # mul: modifica in-place instanta
>>> print(a)
[ThirdClass: abcabcabc]
```



- expresia **a +** apeleaza metoda **__add__()**

A...

- Metodele speciale, ca __init__, __add__, __str__, sunt mostenite de subclase si instante, la fel ca orice metoda, si sunt apelate automat
  - Metoda __**init**__ poate fi apelata si manual pentru initializarea superclasei
- Metodele pot returna rezultate, e.g. __str__
  - __add__ returneaza o instanta noua a lui *ThirdClass*, unde __init__ este apelat ca constructor al noii instante – ce are toate metodele clasei
- Alte metode, e.g. *mul*, fac doar modificari in-place
  - Versiunea cu __**mul**__ ar trebui sa returneze o instanta noua a lui *ThirdClass* (ca __**add**__)

Rolul inlocuirii operatorilor

- Proiectarea claselor se poate face cu sau fara *operator overloading*
  - Operatorii lipsa sau nemosteniti de la vreo superclasa produc exceptii la executia expresiei in care apar
    - *print* afiseaza o valoare implicita
- Metode ce inlocuiesc operatori au sens in prelucrari matematice, vectori/matrici, pentru adunare sau inmultire
  - N-au semnificatie intr-o baza de date de angajati
- Au sens daca clasa este folosita pentru operatii specifice tipurilor predefinite (list, dict)
- **__init__** este importanta, ades folosita
  - initializeaza toate atributele instantei

Cea mai simpla clasa

- Mostenirea se bazeaza pe cautarea atributelor dintr-un arbore de obiecte inlantuite:

```
>>> class rec: pass                    # Obiect, spatiu de nume vid
```

- Adaugare de atribute la obiectul *rec*:

```
>>> rec.name = 'Bob'                   # Obiecte cu atribute
>>> rec.age = 40
```

- Folosire ca record (Pascal) sau struct (C):

```
>>> print(rec.name)
Bob
```

- Creare de instante:

```
>>> x = rec() # Instantele mostenesc numele din clase
>>> y = rec()
>>> x.name, y.name # name este memorat numai in clasa
('Bob', 'Bob')
```

Cea...

- Instantele au mostenit atributul *name*, dar asignarile dintr-o instanta creeaza/modifica atributul instantei – care nu mai este cautat pe lantul de mosteniri:

```
>>> x.name = 'Sue'                    # name din x!
>>> rec.name, x.name, y.name          # y.name este mostenit din rec.name
('Bob', 'Sue', 'Bob')                 # x.name este al instantei x
```

- Dictionarul spatiului de nume al clasei/instantei este **__dict__**:

```
>>> list(rec.__dict__.keys())
['__module__', '__dict__', '__weakref__',
    '__doc__', 'name', 'age']
>>> list(name for name in rec.__dict__ if not
    name.startswith('__')) # rec are 2 atribute
['name', 'age']
>>> list(x.__dict__.keys()) # x are atr. name
['name']
>>> list(y.__dict__.keys()) # y nu are atribute
[]
```

Cea...

- Atributul calificat este cautat pe lantul de mosteniri, insa indexarea lui __dict__ cu stringul nume de atribut este limitata la spatiul de nume curent:

```
>>> x.name, x.__dict__['name'] # Atributele
    sunt cheile dictionarului ca stringuri
('Sue', 'Sue')
>>> x.age # Calificarea cauta si in clasa
40
```

```
>>> x.__dict__['age'] # Indexarea cu cheia 'age'
    nu este gasita – fara mostenire
KeyError: 'age'
```

- Instantele au atributul __**class**__ care refera clasa din care fac parte:

```
>>> x.__class__        # Legatura/referinta de la instanta la clasa
<class '__main__.rec'>
```

Cea...

- Clasele au atributul **__bases__** care este un *tuple* de referinte catre superclasele lor:

```
>>> rec.__bases__     # tuple in v3.x, unde exista superclasa object
(<class 'object'>,)
```

- Si metode pot fi adaugate unei clase:

```
>>> def uppername(obj):  # Cu un argument!
        return obj.name.upper()
>>> uppername(x) # Apel ca functie
'SUE'
>>> rec.method = uppername # Adaugare ca atribut al clasei
>>> x.method() # Apel de metoda a instantei x
'SUE'
>>> y.method() # Apel de metoda a instantei y
'BOB'
>>> rec.method(x) # Apel echivalent, ca metoda a clasei
'SUE'
```

Clase vs. dictionare

- Memorarea informatiilor se poate face cu inregistrari de tip *tuple*, *dict*:

```
>>> rec = ('Bob', 40.5, ['dev', 'mgr']) # Cu tuple
>>> print(rec[0])
Bob
>>> rec = {} # Cu dict
>>> rec['name'] = 'Bob'
>>> rec['age'] = 40.5
>>> rec['jobs'] = ['dev', 'mgr']
>>> rec = dict( name='Bob', age=40.5,
      jobs=['dev', 'mgr']) # Tot cu dict()
>>> print(rec['name'])
Bob
```

- Mai simplu, cu *class*:

```
>>> class rec: pass # Cu class
>>> rec.name = 'Bob'
>>> rec.age = 40.5
>>> rec.jobs = ['dev', 'mgr']
>>> print(rec.name)
Bob
```

Clase...

- Mai bine, cu instante ale aceleiasi clase:

```
>>> class rec: pass # Doar o instructiune de tip
    class e necesara, pentru oricate persoane
>>> pers1 = rec() # Cu instante
>>> pers1.name = 'Bob'
>>> pers1.jobs = ['dev', 'mgr']
>>> pers1.age = 40.5
>>> pers2 = rec()
>>> pers2.name = 'Sue'
>>> pers2.jobs = ['dev', 'cto']
>>> pers1.name, pers2.name
('Bob', 'Sue')
```

- Cu clase complete, cu metode, ceea ce *dict* nu poate:

```
>>> class Person:
        def __init__(self, name, jobs,
    age=None): # class = data + metode
            self.name = name
            self.jobs = jobs
            self.age = age
        def info(self): # Returneaza tuplu!
            return (self.name, self.jobs)
>>> rec1 = Person('Bob', ['dev', 'mgr'], 40.5) #
    Apel de constructor
>>> rec2 = Person('Sue', ['dev', 'cto'])
>>> rec1.jobs, rec2.info() # Atribut si metoda
(['dev', 'mgr'], ('Sue', ['dev', 'cto']))
```

Clase...

- Dar cu **namedtuple**?:

```
>>> from collections import namedtuple
>>> Rec = namedtuple('Rec', ['name', 'age', 'jobs']) # Rec este o clasa (generata)
>>> bob = Rec('Bob', age=40.5, jobs=['dev', 'mgr']) # Cu namedtuple
```

```
>>> rec = {'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']} # Cu dict.
```

- Desi dictionarele si tuplele sunt flexible, clasele permit adaugarea de metode ce implementeaza comportamentul respectivelor obiecte.

Sumar

- Programarea orientata obiect, OOP
- Programarea cu clase in Python
- ***Exemplu pentru seminar***

Clasele *Person* si *Manager*

- Cele doua clase vor fi proiectate, cu instante se va testa functionalitatea, iar cu biblioteca **shelve** – o baza de date orientata obiect, instantele vor fi salvate/restaurate de pe disc.

1 – Crearea instantelor

- Editarea fisierului **person.py** (cu notepad):

```
# Fisierul person.py:
class Person: # Antetul clasei
```

- Adaugarea constructorului:

```
class Person:
    def __init__(self, name, job, pay):      # Constructor cu trei argumente (in afara de self)
        self.name = name                      # Completarea atributelor de instanta
        self.job = job
        self.pay = pay                        # self este noua instanta
```

- Versiune de __init__ cu *job* si *pay* cu valori implicite:

```
class Person:
    def __init__(self, name, job=None, pay=0):
    # Ca orice functie...
        self.name = name
        self.job = job
        self.pay = pay
```

1...

- Testare – incrementala:

*# Fisierul **person.py**, cu cod de test la urma:*

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
```

```
bob = Person('Bob Smith') # Instanta
sue = Person('Sue Jones', job='dev',
      pay=100000) # Alta instanta
print(bob.name, bob.pay) # Afisare atribute
print(sue.name, sue.pay) # Atribute diferite
```

```
C:\code> py -3 person.py
Bob Smith 0
Sue Jones 100000
```

1...

- Cod de test executat doar ca script principal:

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__': # Ca script
    # autotestare
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
C:\code> py -3 person.py
Bob Smith 0
Sue Jones 100000
```

```
C:\code>py -3
Python 3.7.4 (tags/v3.7.4:e09359112e)
>>> import person
>>> # Fara output!
```

2 – Adaugarea metodelor

- Se implementeaza operatii cu tipuri predefinite, e.g.:

```
>>> name = 'Bob Smith' # str
>>> name.split() # Implicit separatorul este spatiul alb
['Bob', 'Smith']
>>> name.split()[-1] # Numele de familie, in pozitia ultima
'Smith'
```

```
>>> pay = 100000 # int
>>> pay *= 1.10 # Plus 10%, devine float
>>> print('%.2f' % pay) # Afisare cu %.2f, float cu 2 zecimale
110000.00
```

- Includerea operatiilor in codul de test al clasei:

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1]) # Nume de familie
    sue.pay *= 1.10 # Marire de salariu
    print('%.2f' % sue.pay)
```

2...

- Incapsularea logicii ca interfata a obiectului
  - **In metode**
  - Codul este factorizat si se elimina redundantele:

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self): # Metoda, comportamentala
        return self.name.split()[-1] # self este instanta implicita
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent)) # Modificare doar aici!

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName()) # Testarea metodelor
    sue.giveRaise(.10)
    print(sue.pay)
```

3 – Inlocuirea operatorilor

- Afisarea unei instante cu informatii utile ale obiectului
  - **__str__** este preferat de *print* si functia predefinita *str()*
  - **__repr__** este ales implicit (si in modul interactiv)
  - Vom folosi *__repr__*:

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self): # Metoda noua
        return '[Person: %s, %s]' %
    (self.name, self.pay) # str-ul de afisat
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev',
    pay=100000)
    print(bob) # print va folosi __repr__()
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

3...

- Output *py -3 person.py*, unde **print(instanta)** afiseaza stringul returnat de metoda **__repr__**:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```

4 – Specializare cu subclase

- Vom demonstra personalizare/mostenire intr-o subclasa a clasei *Person*, numita *Manager*, care redefineste metoda *giveRaise* asa incat se va acorda in plus un bonus:

```
class Manager(Person):                          # Mosteneste atributele lui Person
    def giveRaise(self, percent, bonus=.10): ...  # Redefineste/personalizeaza
```

- Implementare **gresita**, de tip **copiere**: ☺

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):      # Rescrierea, desi corecta, nu factorizeaza codul!
        self.pay = int(self.pay * (1 + percent +
    bonus))
```

4...

- Versiunea corecta, foloseste apelul de metoda a (super)clasei:
  - *instanta.metoda(args...)*
  - ***class.metoda(instanta, args...)***

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)      # Refoloseste codul original, nu il rescrie
                                                     # self.giveRaise() e gresit, ciclu recursiv infinit!!
```

4...

- Ambele clase in acelasi modul:

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
            return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' %
(self.name, self.pay)
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
            Person.giveRaise(self, percent +
bonus)
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev',
pay=100000)
    print(bob); print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)
    tom.giveRaise(.10) # Metoda redefinita
    print(tom.lastName()) #Mostenita
    print(tom)  # Foloseste metoda __repr__
mostenita de la Person
```

4...

- Output *py -3 person.py*:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

- **Polimorfism** – metoda *giveRaise()* produce rezultate care depind de obiectul/instanta care o executa:
  - Managerul incaseaza in plus un bonus de 10%
- Subclasa *Manager* poate sa si **extinda** codul cu o metoda noua, inexistenta in superclasa *Person*

4...

- Modelul **OOP** – puternic:
  - Am folosit mostenirea – nu am rescris clasa *Manager*
  - Am eliminat redundanta care inseamna efort suplimentar de intretinere a codului in viitor

5 – Specializarea constructorilor

- Vom redefini constructorul **__init__** al clasei *Manager* si vom refolosi constructorul superclasei (un manager ar trebui sa aiba deja job-ul *'mgr')*:

```
class Manager(Person):
    def __init__(self, name, pay):                  # Constructor redefinit
        Person.__init__(self, name, 'mgr', pay)     # Apelul constructorului de superclasa
    def giveRaise(self, percent, bonus=.10):        # Restul, la fel
        Person.giveRaise(self, percent + bonus)     # Output identic.
```

5...

- Modelul **OOP** folosit la:
  - Crearea instantelor
  - Adaugarea metodelor/interfete
  - Inlocuire operatori
  - Specializarea metodelor din subclase
  - Specializarea constructorilor

5...

- Combinarea claselor prin **compozitie**:
  - Versiune a clasei *Manager* care include o instanta a clasei *Person* ca *atribut* al fiecarei noi instante de *Manager*

```
class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay) # Instanta a Person, person, inclusa
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus) # Delegatie catre person, intercepteaza atributul giveRaise
    def __getattr__(self, attr): # __getattr__, pentru operatorul selectie de atribut
        return getattr(self.person, attr) # Delegatie, toate celelalte atribute se preiau de la person, cu functia predefinita getattr()
    def __repr__(self):
        return str(self.person) # Metoda inlocuita care returneaza rezultatul functiei predefinite str()
# Restul, la fel
```

6 – Introspectie

- **Instrumentele introspective** permit accesul la elemente ale structurii obiectelor, cu metode si atribute speciale:
  - Atributul de instanta __**class**__ refera clasa din care a fost creata instanta
  - Numele clasei se afla in atributul __**name**__ (ca modulele)
  - Superclasele se afla in atributul de clasa __**bases**__, tuplu
  - Atributul de obiect __**dict**__ este dictionarul atributelor (chei) atasate obiectului
    - *dict* este iterabil!
- Pot fi folosite la corectarea claselor Person si Manager:
  - tom ar trebui sa afiseze Manager, nu Person
  - atribute adaugate in viitor nu vor fi afisate de *__repr__*

6...

- Investigare interactiva:

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> bob # Interactiv se afiseaza cu __repr__
[Person: Bob Smith, 0]
>>> print(bob) # print foloseste __str__ sau
    __repr__
[Person: Bob Smith, 0]
>>> bob.__class__ # Clasa instantei bob
<class 'person.Person'>
>>> bob.__class__.__name__ # Numele clasei
    lui bob
'Person'
>>> list(bob.__dict__.keys()) # Atributele sunt
    cheile lui __dict__
['pay', 'job', 'name']
>>> for key in bob.__dict__:
        print(key, '=>', bob.__dict__[key]) #
    Indexare manuala
pay => 0
job => None
name => Bob Smith
>>> for key in bob.__dict__:
        print(key, '=>', getattr(bob, key)) #
    Cu functia predefinita getattr(ob, numeatr)
pay => 0
job => None
name => Bob Smith
```

6...

- Instrument care afiseaza atributele unei instante:

```
# Fisier classtools.py (nou):
"Instrumente de introspectie"
class AttrDisplay:
    """
Implementeaza/inlocuieste metoda __repr__
ca sa afiseze numele clasei instantei si lista
atributelor sale
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key,
getattr(self, key)))
        return ', '.join(attrs)
    def __repr__(self):
        return '[%s: %s]' %
        (self.__class__.__name__,
        self.gatherAttrs())
if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
    class SubTest(TopTest): pass
    x, y = TopTest(), SubTest()
    print(x)
    print(y)
```

6...

- Clasa *AttrDisplay* este mostenita, furnizeaza *__repr__* pentru *print()*
- Clasa *SubTest* mosteneste pe *__init__* de la superclasa *TopTest*, deci codul din __init__ este executat de doua ori

```
C:\code>py -3 classtools.py                    [SubTest: attr1=2, attr2=3]
[TopTest: attr1=0, attr2=1]
```

- Versiune recursiva, afiseaza si atributele superclaselor:

```
class AttrDisplay:
    def _gatherAttrs(self): # Nu este listata, incepe cu _
        setattrs = set()
        def getatr( ob ): # Functie inclusa
            for key in ob.__dict__:
                if not key.startswith( "_" ): # Doar atributele normale
                    setattrs.add( '%s=%s' % (key, getattr(ob, key)) )
    def scancb( ob ): # Recursiva, scaneaza clasa si superclasele sale
        getatr( ob )
        for c in ob.__bases__: # Tuplu de superclase
            scancb(c)  # Apel recursiv
    getatr( self ) # Instanta
    scancb( self.__class__ ) # Clasa
    return ', '.join(k for k in setattrs)
```

6...

- Autotest:

```
C:\code>py -3 classtools-recursiva.py
[TopTest: count=4, attr2=1, attr1=0]
[SubTest: attr2=3, count=4, attr1=2]
```

- "Ascunderea" numelor de metode in clase se poate face prin prefixare cu:
  - Un singur underscore _ e.g. **_gatherAttrs**
  - Doi underscore __ e.g. **_AttrDisplay__gatherAttrs**: rezulta un nume unic prin combinatie cu numele clasei, pseudo privatizare.
- Versiunea finala a claselor Person si Manager:

*# Fisier **person.py** (final):*

"Baza de date de persoane"

from classtools import AttrDisplay *# Importare instrument de inspectare*

class Person(AttrDisplay): *# Mosteneste __repr__ de la AttrDisplay !!*

 # __repr__ lipseste

 # Restul la fel

7 – Salvarea obiectelor dintr-o baza de date

- Permanentizarea obiectelor se poate face cu:
  - **pickle** – serializeaza obiecte Python din/catre un string de bytes
  - **dbm** – implementeaza un system de fisiere accesibil prin cheie pentru stocarea stringurilor
  - **shelve** – foloseste modulele de mai sus pentru stocarea obiectelor cu cheie
- Modulul *pickle* salveaza orice obiect, chiar si clase
- Modulul *shelve* converteste un obiect cu *pickle* si apoi stocheaza respectivul string intr-un fisier *dbm*, cu o cheie.
  - *shelve* se comporta ca un dictionar, fiindca mapeaza operatiile *dict*-ului cu obiecte dintr-un fisier
  - in plus fata de *dict*, *shelve* trebuie deschis si inchis la sfarsit

7...

- Salvarea obiectelor intr-o baza de date tip **shelve**:

```
# Fisierul makedb.py: stocheaza obiecte de tip
    Person intr-un DB shelve

from person import Person, Manager #
    Incarcare clase

bob = Person('Bob Smith') # Crearea obiectelor
    de stocat

sue = Person('Sue Jones', job='dev',
    pay=100000)

tom = Manager('Tom Jones', 50000)

import shelve

db = shelve.open('persondb') # Deschidere, se
    salveaza in fisierul persondb.*

for obj in (bob, sue, tom): # Iterare per obiectele
    de salvat

    db[obj.name] = obj # Stocare prin indexare
    cu un str

db.close() # Inchidere la incheierea stocarii
```

- Inspectarea interactiva a fisierelor *persondb.**:

```
>>> import glob # Modul care listeaza fisiere
>>> glob.glob('person*')
['person-composite.py', 'person-department.py',
    'person.py', 'person.pyc',
'persondb.bak', 'persondb.dat', 'persondb.dir']
```

7...

```
>>> print(open('persondb.dir').read()) # Citirea intregului fisier, *.dir sunt text
'Sue Jones', (512, 92)
'Tom Jones', (1024, 91)
'Bob Smith', (0, 80)
```

```
>>> print(open('persondb.dat','rb').read()) # Fisier binar:
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00jobq\x03NX\x03\x00
...etc...
```

- Inspectarea interactiva cu *shelve*:

```
>>> import shelve
>>> db = shelve.open('persondb') # Redeschidere
>>> len(db) # Trei inregistrari
3
>>> list(db.keys()) # Cheile
['Sue Jones', 'Tom Jones', 'Bob Smith']
```

```
>>> bob = db['Bob Smith'] # Accesare bob via cheia sa
>>> bob # Se executa __repr__ din AttrDisplay!!
[Person: job=None, name=Bob Smith, pay=0]
>>> bob.lastName() # Se apeleaza metoda lastName a clasei Person
'Smith'
```

7...

```
>>> for key in db: # Iteratie, indexare, print
        print(key, '=>', db[key])
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
```

```
>>> for key in sorted(db):
        print(key, '=>', db[key]) # Iteratie pe cheile sortate
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

- Mecanismul functioneaza bazat pe faptul ca:
  - *shelve* stocheaza/pickle instanta clasei, atributele instantei, numele clasei din care instanta a fost creata si modulul clasei
  - la restaurare, *bob* este extras cu *shelve*, unpickled, clasa este reimportata si *bob* este asociat cu clasa
  - importarea clasei nu mai este necesara, cu exceptia crearii de instante noi

7...

- Actualizarea obiectelor de pe *shelve*:

```
# Fisier updatedb.py: actualizeaza un Person in DB
import shelve
db = shelve.open('persondb') Redeschidere cu acelasi nume de fisier
for key in sorted(db): # Iteratie pentru afisarea obiectelor
    print(key, '\t=>', db[key]) # Afisare cu un format
sue = db['Sue Jones'] # Acces prin indexare
sue.giveRaise(.10) # Actualizare in memorie cu metoda clasei, giveRaise
db['Sue Jones'] = sue # Asignare cu cheia 'Sue Jones' pentru actualizarea din shelve
db.close() # Inchiderea bazei de date – schimbarile devin permanente
```

```
C:\code> py -3 updatedb.py
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

```
C:\code> py -3 updatedb.py ...etc...
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

7...

- Alte instrumente disponibile in Python:
  - GUI cu **tkinter**
  - Constructia de pagini web cu:
    - Django, TurboGears, Pylons, web2Py, Zope, Google APS
  - Servicii web, cu SOAP, XML-RPC
  - Baze de date cu:
    - ZODB (un OODB), relationale SQL – MySQL, Oracle, PostgreSQL, SQLite
    - sunt suportate operatii concurente
  - ORM – Object-Relational Mappers: SQLObject, SQLAlchemy