Programarea calculatoarelor si limbaje de programare II

Proiectarea cu clase

Universitatea Politehnica din Bucureşti

Sumar

- ***Mostenirea in OOP***
- Compozitia in OOP
- Persistenta claselor
- Atribute pseudo-private
- Metode – ca obiecte
- Clase – ca obiecte
- Mostenirea multipla

OOP in Python

- OOP se bazeaza pe:
  - Mostenire – cautarea atributelor in expresii de tip *X.atribut*
  - Polimorfism – executia metodei din expresia *X.metoda* depinde de tipul (clasa) lui *X*
  - Incapsulare – datele/codul sunt ascunse in clase care au un comportament determinat de metodele si operatorii lor
- **Polimorfismul** – se refera la *interfata* suportata de obiecte, nu la tipul datelor:
  - Metode cu acelasi nume dar semnaturi diferite sunt sintactic posibile, insa numai ultima definitie este pastrata
  - Testarea tipului argumentelor, desi posibila, nu este in spiritul programarii in Python
    - se recomanda programarea la nivel de interfata:

      def meth(self, x): x.operatie()

Mostenirea in OOP

- Sintactic, mostenirea este bazata pe calificarea atributelor – care sunt cautate in instanta, clasa si superclase
- Mostenirea se refera la apartenenta la un set/multime de proprietati, definite intr-o clasa, care pot fi mostenite si particularizate in subclase
- Fie exemplul din modulul *employees.py*, cu patru clase, *Employee*, *Chef*, *Server*, *PizzaRobot*:
  - De la *Employee* se mostenesc metodele *giveRaise*, *work* si *__repr__*
  - *Chef* si *Server* mostenesc pe *Employee*, schimba metoda *work*
  - *PizzaRobot* mosteneste pe *Chef*, schimba metoda *work*

Mostenirea...

```
# Fisier employees.py:
from __future__ import print_function
class Employee:
    def __init__(self, name, salary=0):
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary +=  self.salary * percent
    def work(self):
        print(self.name, "does stuff")
    def __repr__(self):
        return "<Employee: name=%s, salary=%.0f>" % (self.name, self.salary)
class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "makes food")
class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, "interfaces with customer")
class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, "makes pizza")
```

Mostenirea...

```
if __name__ == "__main__": #Autotestare
    bob = PizzaRobot('bob') # Se creeaza un robot numit bob
    print(bob) # Afisare cu __repr__, mostenita
    bob.work() # Apel de work, specific
    bob.giveRaise(0.20) # Marire de 20%
    print(bob); print()
```

```
for klass in Employee, Chef, Server, PizzaRobot: # klass este, pe rand, clasa
    obj = klass(klass.__name__) # Argument str, rezultat instanta
    obj.work()
```

```
C:\code>py -3 employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000>
```

```
Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

Sumar

- Mostenirea in OOP
- ***Compozitia in OOP***
- Persistenta claselor
- Atribute pseudo-private
- Metode – ca obiecte
- Clase – ca obiecte
- Mostenirea multipla

Compozitia in OOP

- Sintactic, compozitia presupune includerea de obiecte intr-o clasa/container si folosirea lor la implementarea metodelor clasei
- Proiectarea bazata pe compozitie se refera la componentele containerului
  - Clasa compusa din mai multe obiecte incluse are de obicei o interfata proprie care este implementata prin redirectare catre obiectele continute. Este si un *controller*.
- Fie exemplul din *pizzashop.py*, cu trei clase, *Customer*, *Oven* si *PizzaShop* – compusa din clasele *Server*, *PizzaRobot* si *Oven*:
  - In metoda *PizzaShop.order* se activeaza si un obiect *Customer*

Compozitia...

```
# Fisier pizzashop.py:
from __future__ import print_function
from employees import PizzaRobot, Server
class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, "orders from", server)
    def pay(self, server):
        print(self.name, "pays for item to", server)
class Oven:
    def bake(self):
        print("oven bakes")
```

```
class PizzaShop: # Compozitie
    def __init__(self):
        self.server = Server('Pat') # Incluziune de obiecte
        self.chef = PizzaRobot('Bob') # Inclus si un robot numit bob
        self.oven = Oven() # Inclus si un Oven
    def order(self, name):
        customer = Customer(name) # Activare a altor obiecte – Customer
        customer.order(self.server) # Customer comanda de la server-ul pizzeriei
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)
```

Compozitia...

```
if __name__ == "__main__": # Autotestare
    scene = PizzaShop() # Creare obiect compus
    scene.order('Homer') # Simularea comenzii lui Homer
    print('... ')
    scene.order('Shaggy') # Simularea comenzii lui Shaggy
```

```
C:\code>py -3 pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

- Fie exemplul din *streams.py* si *converters.py*, cu doua clase, *Processor* si respectiv *Uppercase*:
  - *Processor* este compus cu obiectele *reader* si *writer*
  - *Uppercase* mosteneste pe *Processor* si furnizeaza implementarea pentru metoda *converter*

Compozitia...

*# Fisier **streams.py**:*

```
class Processor: # Compozitie
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while True:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
        assert False, 'converter must be defined' # Metoda abstracta, necesita implementare in subclasa
```

*# Fisier **converters.py**:*

```
from streams import Processor
class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':  # Autotestare
    import sys
    print(open('trispam.txt').read())
    print('***After***')
    obj = Uppercase( open('trispam.txt'),
                     sys.stdout
                   )
    obj.process()
```

Compozitia...

```
C:\code>py -3 converters.py
spam
Spam
SPAM!
```

```
***After***
SPAM
SPAM
SPAM!
```

- Cu o clasa, *HTMLize*, care implementeaza *write*:

```
C:\code>py -3
>>> from converters import Uppercase
>>> class HTMLize:
        def write(self, line):
            print('<PRE>%s</PRE>' %
    line.rstrip())
```

```
>>> Uppercase(open('trispam.txt'),
    HTMLize()).process()
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
>>>
```

- Observatie: mostenirea furnizeaza conversia la litere mari iar compozitia ofera formatarea HTML

Sumar

- Mostenirea in OOP
- Compozitia in OOP
- ***Persistenta claselor***
- Atribute pseudo-private
- Metode – ca obiecte
- Clase – ca obiecte
- Mostenirea multipla

Persistenta claselor

- Obiectele pot fi salvate/restaurate de/pe disc, cu modulele *pickle* sau *shelve*:

```
import pickle
object = SomeClass()
file = open(filename, 'wb') # Creare fisier disc
pickle.dump(object, file) # Salvare in fisier
```

```
import pickle
file = open(filename, 'rb')
object = pickle.load(file) # Restaurare obiect
```

```
import shelve
object = SomeClass()
dbase = shelve.open(filename)
dbase['key'] = object # Salvare cu cheia 'key'
```

```
import shelve
dbase = shelve.open(filename)
object = dbase['key'] # Restaurare, acces cu cheie
```

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Employee: name=Pat, salary=40000>,
 <Employee: name=Bob, salary=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.pkl', 'wb'))
```

Persistenta...

```
>>> import pickle
>>> obj = pickle.load(open('shopfile.pkl', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>,
    <Employee: name=Bob, salary=50000>)
>>> obj.order('DSA')
DSA orders from <Employee: name=Pat,
    salary=40000>
Bob makes pizza
oven bakes
DSA pays for item to <Employee: name=Pat,
    salary=40000>
```

- Observatie: obiectele restaurate retin atat starea cat si comportamentul (metodele)

Delegatia in OOP

- Delegatia este un caz special de compozitie:
  - Un singur obiect este inclus intr-o clasa de tip *proxy/wrapper*
    - Interfata clasei este cea a obiectului inclus
- Exemplul cu operatorul inlocuit *__getattr__* si *__repr__*:

```
# Fisier trace.py:
class Wrapper:
    def __init__(self, object):
        self.wrapped = object # Includere obiect, in instanta!
    def __getattr__(self, attrname):
        print('Trace: ' + attrname) # Trasare cu print; attrname este str
        return getattr(self.wrapped, attrname) # Delegatie catre obiectul inclus
    def __repr__(self):
        return str(self.wrapped) # Alta delegatie
```

```
>>> x = Wrapper([1, 2, 3])
>>> x.append(4); x # x este afisat cu __repr__
Trace: append
[1, 2, 3, 4]
```

```
>>> x = Wrapper({'a': 1, 'b': 2})
>>> list(x.keys())
Trace: keys
['a', 'b']
```

Sumar

- Mostenirea in OOP
- Compozitia in OOP
- Persistenta claselor
- ***Atribute pseudo-private***
- Metode – ca obiecte
- Clase – ca obiecte
- Mostenirea multipla

Atribute pseudo-private

- Prefixarea atributelor cu un singur *underscore*, e.g. **_X**, este o simpla conventie
- Atribute care incep cu *doua underscore* si nu se incheie cu dublu *underscore*, e.g. **__X** din clasa *Spam*, vor include, automat, numele clasei, precedat de un singur underscore: **_Spam__X**
  - **Transformarea** se produce pentru toate atributele clasei sau instantei:
    - Metoda *__meth* devine *_Spam__meth*
    - Atributul de instanta *self.__X* devine *self._Spam__X*
- Scopul folosirii atributelor pseudo-private:
  - Deoarece atributele de instanta se afla intr-un **singur obiect** de tip instanta, indiferent de cate superclase exista, este posibila coliziunea de nume

Atribute...

- Exemplu, coliziune pe atributul **X**:

```
class C1:
    def meth1(self): self.X = 88 # X este al meu?!
    def meth2(self): print(self.X)
class C2:
    def metha(self): self.X = 99 # Idem
    def methb(self): print(self.X)
class C3(C1, C2): pass
I = C3() # Doar un singur X in instanta I
# Valoarea lui X depinde de ordinea apelurilor lui meth1 si metha
```

- Atributul va apartine clasei, daca este prefixat cu **__X**:

```
# Fisier pseudoprivate.py:
class C1:
    def meth1(self): self.__X = 88 # X al clasei
    def meth2(self): print(self.__X) # Devine _C1__X in I
class C2:
    def metha(self): self.__X = 99 # Idem
    def methb(self): print(self.__X) # Devine _C2__X in I
class C3(C1, C2): pass
I = C3() # Doua nume X in I
I.meth1(); I.metha()
print(I.__dict__)
I.meth2(); I.methb()
```

Atribute...

- Output:

```
C:\code>py -3 pseudoprivate.py
{'_C1__X': 88, '_C2__X': 99}
88
99
```

- Accesul din exterior este in continuare posibil e.g. **I._C1__X = 77**
- Evitarea coliziunii pentru o metoda ce trebuie folosita doar in clasa in care a fost definita:

```
class Super:
    def method(self): ...  # O metoda reala
class Tool:
    def __method(self): ...  # Devine _Tool__method
    def other(self): self.__method()  # Foloseste metoda proprie
class Sub1(Tool, Super): ...
    def actions(self): self.method()  # Se executa Super.method
class Sub2(Tool):
    def __init__(self): self.method = 99  #  Tool.__method este neafectata!
```

Sumar

- Mostenirea in OOP
- Compozitia in OOP
- Persistenta claselor
- Atribute pseudo-private
- ***Metode – ca obiecte***
- Clase – ca obiecte
- Mostenirea multipla

Metode – ca obiecte, legate sau simple

- Metodele sunt obiecte, exista doua categorii:
  1. Medode (de clasa) **simple** – fara *self*
     - in v3.x, o metoda simpla este o functie apelabila prin calificarea cu clasa
  2. Metode (de instanta) **legate** – cu *self* + functie
     - instanta si functia sunt impachetate intr-un obiect de tip metoda legata (iar instanta nu se mai transmite ca prim argument la apel)

- Exemplu:

```
class Spam:
    def doit(self, message):
        print(message)
```

```
object1 = Spam()
object1.doit('hello world') # Apel normal
```

- Apel cu metoda legata:

```
object1 = Spam()
x = object1.doit # Metoda legata, x:
    instanta+functie
```

```
x('hello world') # Apel, totuna cu
    object1.doit('...')
```

Metode...

- Apel cu metoda simpla:

```
object1 = Spam()
t = Spam.doit # Metoda simpla, t
t(object1, 'howdy') # Cu prim argument instanta
    (daca are self, in v3.X)
```

- Expresia *self.metoda* este o metoda legata:

```
class Eggs:
    def m1(self, n):
        print(n)
    def m2(self):
x = self.m1 # x, metoda legata
x(42) # Apel, ca o functie obisnuita
Eggs().m2() # Afiseaza 42
```

- Metodele simple sunt functii obisnuite in Python v3.x
  - In v3.x **metodele calificate cu clasa** pot fi apelate fara instanta, daca *self* nu este prevazut in definitie

Metode...

- Exemplu, v3.x:

```
>>> class Selfless:
        def __init__(self, data):
            self.data = data
        def selfless(arg1, arg2): # O functie obisnuita in 3.X
            return arg1 + arg2
        def normal(self, arg1, arg2): # Instanta este necesara la apel
            return self.data + arg1 + arg2
```

```
>>> X = Selfless(2)
>>> X.normal(3, 4) # Apel normal, self este transmis automat: 2+(3+4)
9
>>> Selfless.normal(X, 3, 4) # self este primul argument – la calificarea cu clasa
9
>>> Selfless.selfless(3, 4) # Fara instanta: merge in 3.X (nu in 2.X!)
7
```

- Erori, in v3.x si v2.x:

```
>>> X.selfless(3, 4) # self este pasat automat la apelul prin instanta
TypeError: selfless() takes 2 positional arguments but 3 were given
```

Metode...

```
>>> Selfless.normal(3, 4) # Interpreteaza primul argument ca fiind self, constata lipsa lui arg2
TypeError: normal() missing 1 required positional argument: 'arg2'
```

- Metode legate si alte obiecte apelabile
  - Metodele legate pot fi procesate ca obiecte generice, apelabile:

```
>>> class Number:
        def __init__(self, nr):
            self.nr = nr
        def double(self):
            return self.nr * 2
        def triple(self):
            return self.nr * 3
>>> x = Number(2) # Instanta
>>> y = Number(3) # Instanta, cu alta stare, 3
>>> z = Number(4)
>>> x.double() # Apel normal
4
>>> acts = [x.double, y.double, y.triple, z.double] # Lista cu metode legate
>>> for act in acts: # Apel intarziat
        print(act(), end=' ') # Apelate ca functii obisnuite
4 6 9 8
```

Metode...

- Informatii (atribute) ale metodelor legate: ***__self__*** si ***__func__***:

```
>>> bound = x.double # Metoda de instanta, legata
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x000001C333B29088>, <function Number.double at 0x000001C333B59948>)
>>> bound.__self__.nr
2
>>> bound() # Echivalent cu apel de bound.__func__(bound.__self__, ...)
4
```

- Alte obiecte apelabile:

```
>>> def square(arg):
        return arg ** 2 # Functie obisnuita (def sau lambda)
>>> class Sum: # Are instante apelabile
        def __init__(self, val):
            self.val = val
        def __call__(self, arg):
            return self.val + arg
```

```
>>> class Product: # method() se va lega
        def __init__(self, val):
            self.val = val
        def method(self, arg):
            return self.val * arg
```

Metode...

```
>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method]
    # Functie, instanta, metoda legata
>>> for act in actions: # Apelabile la fel
        print(act(5), end= ' ') # Cu un argument
25 7 15
>>> actions[-1](5) # Indexare
15
>>> [act(5) for act in actions] # Lista iterabila
[25, 7, 15]
>>> list(map(lambda act: act(5), actions)) # Mapare
[25, 7, 15]
```

- Si clasele sunt apelabile (produc instante):

```
>>> class Negate: # Clasa, apelabila, produce instante!
        def __init__(self, val):
            self.val = -val
        def __repr__(self): # Pentru print
            return str(self.val)
>>> actions = [square, sobject, pobject.method, Negate] # Plus o clasa!
>>> for act in actions:
        print(act(5), end=' ')
25 7 15 -5
```

Metode...

```
>>> [act(5) for act in actions]  # Colectie/Lista iterativa
[25, 7, 15, -5]
>>> table = {act(5): act for act in actions} # Dictionar iterativ, valorile sunt obiecte
>>> for (key, value) in table.items():
        print('%2s => %s' % (key, value))
```

```
25 => <function square at 0x000001C333B59A68>
 7 => <__main__.Sum object at 0x000001C333B2F608>
15 => <bound method Product.method of <__main__.Product object at 0x000001C333B2FD08>>
-5 => <class '__main__.Negate'>
```

- Metode legate se folosesc in proiectarea grafica (*tkinter*):

```
class MyGui:
    def handler(self): ... # Foloseste self.attr pentru memorarea starii
    def makewidgets(self):
        b = Button(text='spam', command=self.handler) # self.handler este metoda legata, cu acces la atributele instantei
```

Sumar

- Mostenirea in OOP
- Compozitia in OOP
- Persistenta claselor
- Atribute pseudo-private
- Metode – ca obiecte
- ***Clase – ca obiecte***
- Mostenirea multipla

Clase – ca obiecte, fabrici de obiecte

- Clasele sunt obiecte in Python
  - Crearea obiectelor la executie, nu proiectare, este posibila cu tehnica producerii obiectelor cunoscuta ca *fabrici* de obiecte
- Exemplu:

```
def factory(aClass, *pargs, **kargs): # Argumente multiple: tuple, dict
    return aClass(*pargs, **kargs) # Apel de aClass (sau cu apply() in v2.X )

class Spam:
    def doit(self, message):
        print(message)

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

object1 = factory(Spam) # O instanta Spam
object2 = factory(Person, "Arthur", "King") # O instanta Person
object3 = factory(Person, name='Brian') # Idem, cu cuvinte cheie si valoare implicita
```

```
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Arthur', 'King')
>>> object3.name, object3.job
('Brian', None)
```

Clase...

- Rolul fabricilor de obiecte:
  - Permit crearea obiectelor la executie
- Exemplu, procesorul care citeste cu o clasa *descrisa* intr-un fisier de configurare:

```
classname = ...string citit din fisier...
classarg = ...argumente citite din fisier...
import streamtypes # Modul cu diverse clase
aclass = getattr(streamtypes, classname) #
    Clasa este atribut din modul
reader = factory(aclass, **classarg) # Sau chiar
    aclass(**classarg)
processor(reader, ...)
```

Sumar

- Mostenirea in OOP
- Compozitia in OOP
- Persistenta claselor
- Atribute pseudo-private
- Metode – ca obiecte
- Clase – ca obiecte
- ***Mostenirea multipla***

Mostenirea multipla – clase mixate

- Mostenirea multipla, de la mai multe (super)clase:
  - Listate intre paranteze(cu virgule), in antetul clasei
- Ordinea mostenirii, de la stanga la dreapta, dar si in adancime:
  - DFLR – Depth-First, Left to Right (apoi) pana la v3.0 inclusiv
  - MRO – Method Resolution Order, in v3.X, optionala in v2.x
    - difera in cazul mostenirii comune – **romb**: intai pe orizontala, apoi in adancime; se cauta in toate clasele derivate din cea comuna, apoi ea.
- Conflictele de nume comune in mai multe clase se rezolva:
  - **Implicit** – cea mai de jos si din stanga arborelui de mostenire, cu exceptia romburilor: cea din dreapta si apoi de mai de sus
  - **Explicit**: *Superclasa.metoda(self)*, deci selectie expresa

Mostenirea...

- Exemplu, listarea atributelor de instanta cu **__dict__**:

```
#!python

# Fisier listinstance.py (v2.X si v3.X):

class ListInstance:
    """
    Mixare de clase care folosesc __str__ pentru
    print() sau str(); se afiseaza doar atributele
    de instanta; pseudoprivatizare cu __X
    """
    def __attrnames(self):
        result = ' '
        for attr in sorted(self.__dict__):
            result += '\t%s=%s\n' % (attr,
self.__dict__[attr])
        return result
    def __str__(self):
        return '<Instance of %s, address
%s:\n%s>' % (
            self.__class__.__name__, # Numele
clasei
            hex( id(self) ), # Adresa, cu functia
predefinita id(obiect), in hexazecimal
            self.__attrnames()) # Listare
numeatr=valoareatr
if __name__ == '__main__': # Autotestare
    import testmixin # Un modul cu doua clase,
Super si Sub(Super, ListInstance)
    testmixin.tester(ListInstance)
```

Mostenirea...

- Versiunea cu o colectie iterativa si *join*:

```
def __attrnames(self):
    return ''.join( '\t%s=%s\n' % (attr, self.__dict__ [attr]) for attr in sorted(self.__dict__)  )
```

```
>>> # Exemplu:
>>> from listinstance import ListInstance
>>> class Spam(ListInstance): # Mosteneste metoda __str__
        def __init__(self):
            self.data1 = 'food'

>>> x = Spam()
>>> print(x) # print() si str() folosesc __str__
<Instance of Spam, address 0x1fd28553808:
    data1=food
>
>>> display = str(x)
>>> display
'<Instance of Spam, address
    0x1fd28553808:\n\tdata1=food\n>'
>>> x # Afisat cu __repr__, aceeasi adresa!
<__main__.Spam object at
    0x000001FD28553808>
>>>
```

Mostenirea...

- Modulul *testmixin.py*:

```
#!python

# Fisierul testmixin.py (v2.X si v3.X)

import importlib # Pentru import_module()

def tester(listerclass, sept=False):

    class Super:

        def __init__(self):

            self.data1 = 'spam'

        def ham(self): pass

    class Sub(Super, listerclass): # Mixare,
    pentru ham si __str__

        def __init__(self): # Acces la self

            Super.__init__(self)

            self.data2 = 'eggs'

            self.data3 = 42

        def spam(self): pass

    instance = Sub() # O instanta cu __str__
    mostenit

    print(instance) # Folosirea lui __str__ de
    catre print()

    if sept: print('-' * 80)

def testByNames(modname, classname,
    sept=False):

    modobject=importlib.import_module(
    modname ) # Argument string

    listerclass = getattr(modobject, classname)
    # Clasa, atribut de modul

    tester(listerclass, sept)
```

Mostenirea...

```
if __name__ == '__main__': # Autotestare
    testByNames('listinstance', 'ListInstance',
    True)
    testByNames('listinherited', 'ListInherited',
    True)
    testByNames('listtree', 'ListTree', False)
```

```
C:\code>py -3  listinstance.py
<Instance of Sub, address 0x22cb1002f08:
    data1=spam
    data2=eggs
    data3=42
>
```

```
C:\code>py -3 testmixin.py
<Instance of Sub, address 0x1fcdb3a0188:
    data1=spam
    data2=eggs
    data3=42
>
---------etc-----
```

```
>>> import listinstance
>>> class C(listinstance.ListInstance): pass
>>> x = C()
>>> x.a, x.b, x.c = 1, 2, 3
```

```
>>> print(x) # Mixare cu orice clasa e OK
<Instance of C, address 0x1fd285537c8:
    a=1
    b=2
    c=3
>
```

Mostenirea...

- Exemplu, listarea atributelor mostenite cu ***dir()***:

```
#!python
# Fisier listinherited.py (v2.X si v3.X):
class ListInherited:
    """
    Cu dir() se colecteaza atat atributele de
    instanta cat si cele mostenite de la clase
    """
    def __attrnames(self): # Pseudoprivata
        result = ''
        for attr in dir(self): # dir() aplicat instantei
            if attr[:2] == '__' and attr[-2:] == '__': # Fara atribute interne
                result += '\t%s\n' % attr
            else:
                result += '\t%s=%s\n' % (attr, getattr(self, attr))
        return result
    def __str__(self):
        return '<Instance of %s, address %s:\n%s>' % (
          self.__class__.__name__, # Nume clasa
          id(self), # Adresa
          self.__attrnames() )
if __name__ == '__main__': # Autotestare
    import testmixin
    testmixin.tester(ListInherited)
```

Mostenirea...

C:\code>**py -2 listinherited.py**

<Instance of Sub, address 51822152:

_ListInherited__attrnames=<bound method Sub.__attrnames of <testmixin.Sub instance at 0x000000000316BE48>>

__doc__

__init__

__module__

__str__

data1=spam

data2=eggs

data3=42

ham=<bound method Sub.ham of <testmixin.Sub instance at 0x000000000316BE48>>

spam=<bound method Sub.spam of <testmixin.Sub instance at 0x000000000316BE48>>

>

C:\code>**rem Vor fi mult mai multe atribute interne (__X__) in v3.x, mostenite de la clasa *object*!**

- Atributele interne se pot lista pe orizontala:

Mostenirea...

```
def __attrnames(self, indent=' '*4):
        result =
'Unders%s\n%s%%s\nOthers%s\n' % (
'-'*77, indent, '-'*77)
        unders = [] # Lista atributelor interne
        for attr in dir(self):
            if attr[:2] == '__' and attr[-2:]
== '__': # Fara attribute interne
                unders.append(attr)
            else:
                display = str(getattr(self,
attr))[:82-(len(indent) + len(attr))]
                result += '%s%s=%s\n' %
(indent, attr, display)
        return result % ', '.join(unders)
```

 # %% devine % asa incat %%s devine %s si nu este substituit! Decat mai jos.

 # Substituire de %s !

Mostenirea...

C:\code>**py -3 listinherited2.py**

<Instance of Sub, address 1592251765064:

Unders----------

__class__, __delattr__, __dict__, __dir__, __doc__, __eq__, __format__, __ge__, __getattribute__, __gt__, __hash__, __init__, __init_subclass__, __le__, __lt__, __module__, __ne__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __sizeof__, __str__, __subclasshook__, __weakref__

Others----------

_ListInherited__attrnames=<bound method ListInherited.__attrnames of <testmixin

data1=spam

data2=eggs

data3=42

ham=<bound method tester.<locals>.Super.ham of <testmixin.tester.<locals>.Sub o

spam=<bound method tester.<locals>.Sub.spam of <testmixin.tester.<locals>.Sub o

>

C:\code>**rem Intr-adevar, v3.x are multe atribute interne...**

C:\code>**rem S-a folosit __str__ si *nu* __repr__ deoarece s-ar produce apeluri recursive infinite**

Mostenirea...

- Exemplu, listarea atributelor per clasa de care apartin:

```
#!python

# Fisier listtree.py (v2.X si v3.X):

class ListTree:
    """
__str__ returneaza arborele de mostenire
    """
    def __attrnames(self, obj, indent):
        spaces = ' ' * (indent + 1)
        result = ''
        for attr in sorted(obj.__dict__):
            if attr.startswith('__') and
attr.endswith('__'):
                result += spaces +
'{0}\n'.format(attr)
            else:
                result += spaces +
'{0}={1}\n'.format(attr, getattr(obj, attr))
        return result
    def __listclass(self, aClass, indent):
        dots = '.' * indent
        if aClass in self.__visited:
            return '\n{0}<Class {1}:,
address {2}: (see above)>\n'.format(
                dots,
                aClass.__name__,
                id(aClass))
        self.__visited[aClass] = True
# Se continua...
```

Mostenirea...

```
        here = self.__attrnames(aClass, indent)
        above = ''
        for super in aClass.__bases__: # Superclasele clasei curente
            above += self.__listclass(super, indent+4) # Apel recursiv!
        return '\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format( # Mai clar cu format()!
            dots,
            aClass.__name__,
            id(aClass),
            here, above,
            dots)
def __str__(self): # Operatorul inlocuit!
        self.__visited = {}
        here = self.__attrnames(self, 0)
        above=self.__listclass(self.__class__, 4)
        return '<Instance of {0}, address {1}:\n{2}{3}>'.format(
            self.__class__.__name__,
            id(self),
            here, above)

if __name__ == '__main__': # Autotestare
    import testmixin
    testmixin.tester(ListTree)
```

Mostenirea...

C:\code>**py -2 listtree.py**

<Instance of Sub, address 57195144:

_ListTree__visited={}

data1=spam

data2=eggs

data3=42

....<Class Sub, address 57084520:

__doc__

__init__

__module__

spam=<unbound method Sub.spam>

.....<Class Super, address 57084424:

__doc__

__init__

__module__

ham=<unbound method Super.ham>

.....>

.....<Class ListTree, address 54794088:

_ListTree__attrnames=<unbound method ListTree.__attrnames>

_ListTree__listclass=<unbound method ListTree.__listclass>

__doc__

__module__

__str__

.....>

....>

>

C:\code>**rem Cu v3.x apare si clasa *object*!**

Mostenirea...

- Exemplu, listarea atributelor unui modul mare:

```
C:\code>py -3
>>> from listtree import ListTree
>>> from tkinter import Button
>>> class MyButton(ListTree, Button): pass
>>> B = MyButton(text='spam')
>>> len(str(B)) # 20K!
20252
>>> B.pack()
>>>
```