Programarea calculatoarelor si limbaje de programare II

Notiuni avansate - clase

Universitatea Politehnica din Bucureşti

Sumar

- ***<u>Extinderea tipurilor predefinite</u>***
- Clase in stil nou
- Extensii ale claselor in stil nou
- Metode statice si de clasa
- Decoratori si metaclase
- Decoratori de functii definiti de utilizator
- Decoratori de clase
- Metaclase
- Functia *super()*
- Proiectarea cu Clase

Extinderea tipurilor predefinite

- Extinderea unui *list* – prin delegatie, cu operatii de *set*:

```
class Set:
    def __init__(self, value = []): # Constructor
        self.data = [] # List, obiect extins
        self.concat(value) # Mai jos, concat()
    def intersect(self, other): # other: secventa
        res = []
        for x in self.data:
            if x in other: # Intersectie
                res.append(x)
        return Set(res) # Un Set nou
    def union(self, other): # other: secventa
        res = self.data[:] # Copiere!
        for x in other:
            if not x in res:
                res.append(x)
        return Set(res)
    def concat(self, value): # value: list, Set...
        for x in value: # Fara duplicate
            if not x in self.data:
                self.data.append(x)
    def __len__(self): return len(self.data) # len(self), if self
    def __getitem__(self, key): return self.data[key] # self[i], self[i:j]
    def __and__(self, other): return self.intersect(other) # self & other
    def __or__(self, other): return self.union(other) # self | other
# Se continua...
```

Extinderea...

def __*repr*__(self): return 'Set:' + repr(self.data) *# print(self),...*	**def __*iter*__(self): return iter(self.data)** *# for x in self,...*

>>> **from setwrapper import Set**
>>> **x = Set([1, 3, 5, 7])**
>>> **print(x.union(Set([1, 4, 7])))**
	Set:[1, 3, 5, 7, 4]
	>>> **print(x | Set([1, 4, 6]))**
	Set:[1, 3, 5, 7, 4, 6]

- <u>Extinderea tipurilor predefinite prin subclasare(derivare):</u>

```
class MyList(list): # O lista de la 1..N!
    def __getitem__(self, offset):
        print('(indexing %s at %s)' % (self, offset))
        return list.__getitem__(self, offset - 1)

if __name__ == '__main__': # Autotestare
    print(list('abc'))
```

```
x = MyList('abc') # __init__ mostenit de la list
print(x) # __repr__ mostenita de la list
print(x[1]) # MyList.__getitem__, redefinita!
print(x[3])
x.append('spam'); print(x) # De la superclasa list
x.reverse(); print(x) # Idem
```

Extinderea...

```
C:\code>py -3 typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(indexing ['a', 'b', 'c'] at 1)
a
(indexing ['a', 'b', 'c'] at 3)
c
['a', 'b', 'c', 'spam']
['spam', 'c', 'b', 'a']
```

- Extinderea unui *list* prin subclasare(derivare):

```
from __future__ import print_function # v2.X
class Set(list):
    def __init__(self, value = []): # Constructor
        list.__init__(self, []) # Apel de constructor al superclasei
        self.concat(value) # Concatenare
    def intersect(self, other): # other: secventa
        res = []
        for x in self:
            if x in other: # Intersectie
                res.append(x)
        return Set(res) # Rezultat: Set nou!
    def union(self, other): # other: secventa
        res = Set(self) # Copiere
        res.concat(other)
        return res
```

Extinderea...

```
def concat(self, value): # value: list, Set, etc.
        for x in value: # Fara duplicate
            if not x in self:
                self.append(x)
def __and__(self, other): return self.intersect(other)
def __or__(self, other): return self.union(other)
def __repr__(self): return 'Set:' + list.__repr__(self)

if __name__ == '__main__': # Autotestare
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print(x, y, len(x))
    print(x.intersect(y), y.union(x))
    print(x & y, x | y)
    x.reverse(); print(x)
```

```
C:\code>py -3 setsubclass.py
Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4
Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]
Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]
Set:[7, 5, 3, 1]
```

Sumar

- Extinderea tipurilor predefinite
- ***Clase in stil nou***
- Extensii ale claselor in stil nou
- Metode statice si de clasa
- Decoratori si metaclase
- Decoratori de functii definiti de utilizator
- Decoratori de clase
- Metaclase
- Functia *super()*
- Proiectarea cu Clase

Schimbari in clasele in stil nou

- In v3.x toate clasele sunt in stil nou
- In v2.x clasele sunt in stil clasic, cu exceptia derivarii dintr-un tip ***predefinit*** sau din clasa ***object*** – cand devin clase in stil nou:
  - *class newstyle(object): ... # Derivare explicita din object in v2.x*
- Schimbari in clasele in stil nou:
  1. Metodele __*getattr*__ si __*getattribute*__ nu mai intercepteaza atribute in urma operatiilor implicite
  2. Tipurile de date sunt clase si clasele sunt tipuri.
  3. Clasele in stil nou mostenesc clasa *object*
  4. Mostenirea multipla se bazeaza pe MRO
  5. Contin: sloturi, proprietati, descriptori, *super* si __getattribute__

1. Operatorii predefiniti __X__ nu mai apeleaza __getattr__/__getattribute__

- Operatorii predefiniti (__*X*__) nu mai apeleaza metodele __*getattr*__ si __*getattribute*__ in mod implicit ci doar explicit:
  - **X[I] este X.__getitem__(I)** in stil clasic
  - **X[I] este type(X).__getitem__(X, I)** in stil nou, cautarea incepe in clasa nu instanta
  - Avantajul noului stil – optimizarea executiei (fara instanta)

```
>>> class C:
    data = 'spam'
    def __getattr__(self, name): # Stil clasic, este apelata de operatorii predefiniti
        print(name)
        return getattr(self.data, name)
```

```
>>> X = C()
>>> X[0]
__getitem__
's'
>>> print(X)
__str__
spam
```

1...

```
>>> class C(object): # Stil nou, mostenire object
        data = 'spam'
        def __getattr__(self, name):
            print(name)
            return getattr(self.data, name)
```

```
>>> X = C() # __getattr__ nu mai este apelata de operatorii predefiniti
>>> X[0]
TypeError: 'C' object does not support indexing
>>> print(X)
<__main__.C object at 0x0000000003E66DC8>
```

- Apelurile explicite de metode __*X*__ sau cu nume normale este posibil:

```
>>> class C: pass # 2.X, stil clasic de clase

>>> X = C()
>>> X.normal = lambda: 99
>>> X.normal()
99
```

```
>>> X.__add__ = lambda y: 88 + y
>>> X.__add__(1)
89
>>> X + 1 # Implicit, merge in stil clasic
89
```

1...

>>> **class C(object): pass** *# Stil nou de clase*
 	
>>> **X = C()**
>>> **X.normal = lambda: 99**
>>> **X.normal()**
99
	>>> **X.__add__ = lambda y: 88 + y**
	>>> **X.__add__(1)** *# Apel explicit*
	89
	>>> **X + 1** *# Apel implicit, expresie*
	TypeError: unsupported operand type(s) for +: 'C' and 'int'

>>> **class C(object):** *# Stil nou*
def __getattr__(self, name): print(name)
 	
>>> **X = C()**
>>> **X.normal** *# Nume normal (nu __X__)*
	normal
	>>> **X.__add__** *# Apel direct*
	__add__
	>>> **X + 1** *# Expresiile nu sunt suportate*
	TypeError: unsupported operand type(s) for +: 'C' and 'int'

- **Clasele de tip proxy (delegatie) sunt affectate**:
  - necesita atat __***getattr***__ (pentru nume normale) cat si **redefinirea** numelor accesate cu operatii predefinite, in expresii.

1...

```
>>> class C(object): # Stil nou
        data = 'spam'
        def __getattr__(self, name):
            print('getattr: ' + name)
            return getattr(self.data, name)
>>> X = C()
>>> X.__getitem__(1)
getattr: __getitem__
'p'
>>> X[1] # Expresiile nu merg
TypeError: 'C' object does not support indexing
>>> X.__add__('eggs')
getattr: __add__
'spameggs'
>>> X + 'eggs' # Nu merge...
TypeError: unsupported operand type(s) for +:
    'C' and 'str'
```

```
>>> class C(object): # Stil nou
        data = 'spam'
        def __getattr__(self, name): # Pentru
    nume normale
            print('getattr: ' + name)
            return getattr(self.data, name)
        def __getitem__(self, i): # Redefinire
            print('getitem: ' + str(i))
            return self.data[i]
        def __add__(self, other): # Redefinire
            print('add: ' + other)
            return getattr(self.data,
    '__add__')(other)
```

1...

```
>>> X = C()
>>> X.upper # Nume normal
getattr: upper
<built-in method upper of str object at
    0x00000000378A4E0>
>>> X.upper()
getattr: upper
'SPAM'
>>> X[1] # Operatie implicita, indexare
getitem: 1
'p'
>>> X.__getitem__(1) # Apel explicit/clasic
getitem: 1
'p'
>>> type(X).__getitem__(X, 1) # Stil nou,
    echivalent
getitem: 1
'p'
>>> X + 'eggs' # La fel pentru adunare:
add: eggs
'spameggs'
>>> X.__add__('eggs')
add: eggs
'spameggs'
>>> type(X).__add__(X, 'eggs')
add: eggs
'spameggs'
```

2. Tipurile de date sunt clase si clasele sunt tipuri

- Tipurile de date sunt clase si clasele sunt tipuri.
  - Obiectul ***type*** genereaza clase ca instante ale sale
  - Clasele genereaza instante proprii

```
>>> # Python v2.x
>>> class C: pass # Stil clasic
>>> I = C()
>>> type(I), I.__class__
(<type 'instance'>, <class __main__.C at
    0x0000000002DA1A68>)
>>> type(C)
<type 'classobj'>
```

```
>>> C.__class__
AttributeError: class C has no attribute
    '__class__'
>>> type([1, 2, 3]), [1, 2, 3].__class__
(<type 'list'>, <type 'list'>)
>>> type(list), list.__class__
(<type 'type'>, <type 'type'>)
```

```
>>> # Python v2.x
>>> class C(object): pass # Stil nou
>>> I = C()
>>> type(I), I.__class__
(<class '__main__.C'>, <class '__main__.C'>)
```

```
>>> type(C), C.__class__
(<type 'type'>, <type 'type'>)
```

2...

```
>>> # Python v3.x:
>>> class C: pass
>>> I = C() # In v3.x, toate clasele sunt in stil nou!
>>> type(I), I.__class__ # Tipul instantei este
      clasa din care provine
(<class '__main__.C'>, <class '__main__.C'>)

>>> type(C), C.__class__ # Clasa este un type, si
      type este o clasa
(<class 'type'>, <class 'type'>)
>>> type([1, 2, 3]), [1, 2, 3].__class__
(<class 'list'>, <class 'list'>)
>>> type(list), list.__class__ # Clasele si tipurile
      predefinite sunt la fel
(<class 'type'>, <class 'type'>)
```

```
# Python v3.x:
def isclass(object):
    return isinstance(object, type)
```

- **Din modulul *inspect***

```
# Python v2.x
def isclass(object):
    return isinstance(object,
                      (type, types.ClassType)
                      )
```

2...

Testarea tipurilor de date:

```
C:\code> py -3
>>> class C: pass # Stil nou de clase
>>> class D: pass
>>> c, d = C(), D()
>>> type(c) == type(d) # In v3.x sunt comparate
      clasele instantelor
False
>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)
>>> c1, c2 = C(), C()
>>> type(c1) == type(c2)
True
```

```
C:\code> py -2
>>> class C: pass # Stil clasic de clase
>>> class D: pass
>>> c, d = C(), D()
>>> type(c) == type(d) # In v2.X toate instantele
      sunt de acelasi tip
True
>>> c.__class__ == d.__class__ # Comparatie
      explicita a claselor
False
>>> type(c), type(d)
(<type 'instance'>, <type 'instance'>)
>>> c.__class__, d.__class__
(<class __main__.C at 0x0000000003464FA8>,
      <class __main__.D at
      0x00000000034292E8>)
```

2...

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

>>> **class C(object): pass** *# <u>Stil nou</u>*

>>> **class D(object): pass**

>>> **c, d = C(), D()**

>>> **type(c) == type(d)** *# La fel ca in v3.X*

False

>>> **type(c), type(d)**

(<class '__main__.C'>, <class '__main__.D'>)

>>> **c.__class__, d.__class__**

(<class '__main__.C'>, <class '__main__.D'>)

- **Verificarea tipurilor nu este recomandata in Python!**

3. Clasele deriva din *object*

- Toate clasele deriva din ***object***:

>>> **class C: pass** *# v3.x* >>> **X = C()**	>>> **type(X), type(C)** *# type este clasa din care a derivat clasa C* (<class '__main__.C'>, <class 'type'>)
>>> **isinstance(X, object)** True	>>> **isinstance(C, object)** *# Clasele mostenesc intotdeauna pe object* True

- Tipurile predefinite sunt clase:

>>> **type('spam'), type(str)**
(<class 'str'>, <class 'type'>)
>>> **isinstance('spam', object)**
True
	>>> **isinstance(str, object)**
	True

- ***type*** deriva din ***object*** si viceversa:

>>> **type(type)** <class 'type'>	>>> **type(object)** <class 'type'>

3...

>>> **isinstance(type, object)** *# Chiar si clasa type deriva din object*

True

>>> **isinstance(object, type)** *# type face clase; type este o clasa*

True

>>> **type is object**

False

- Clasele noi, fiind derivate din *object*, mostenesc atributele acestuia:

>>> **class C: pass**

>>> **C.__bases__**

(<class 'object'>,)

>>> **C().__repr__**

<method-wrapper '__repr__' of C object at 0x000002B1C4398A88>

4. Mostenirea DFLR si MRO

- Mostenirea multipla – sub forma de **romb**:
  - Pentru clasele clasice: DFLR
  - Pentru clasele in stil nou: MRO pentru toate atributele (nu numai metode)
    - se evita vizitarea aceleiasi superclase de mai multe ori
  - Exemple:

>>> **class A: attr = 1** *# Clasic, v2.x*
>>> **class B(A): pass** *# B si C mostenesc de la A*
>>> **class C(A): attr = 2**
>>> **class D(B, C): pass** *# Cauta pe A inainte de C*
	>>> **x = D()**
	>>> **x.attr** *# Cauta in x, D, B, A – unde gaseste pe attr = 1*
	1

>>> **class A(object): attr = 1** *# Stil nou (object necesar doar in v2.x)*
>>> **class B(A): pass**
>>> **class C(A): attr = 2**
>>> **class D(B, C): pass** *# Cauta pe C inainte de A*
	>>> **x = D()**
	>>> **x.attr** *# Cauta x, D, B, C – unde gaseste pe attr = 2*
	2

4...

- Rezolvarea explicita a conflictelor de mostenire:

>>> **class A: attr = 1** # *Stil clasic*
>>> **class B(A): pass**
>>> **class C(A): attr = 2**
>>> **class D(B, C): attr = C.attr** # *<= Se allege C din dreapta*
	>>> **x = D()**
	>>> **x.attr** # *Rezultat ca in stilul nou (v3.x)*
	2

>>> **class A(object): attr = 1** # *Stil nou*
>>> **class B(A): pass**
>>> **class C(A): attr = 2**
>>> **class D(B, C): attr = B.attr** # *<= Se alege A.attr, de deasupra*
	>>> **x = D()**
	>>> **x.attr** # *Rezultat ca in stilul clasic (implicit in v2.x)*
	1

>>> **class A:** # *Cu metode, nu numai atribute*
    def meth(s): print('A.meth')
>>> **class C(A):**
    def meth(s): print('C.meth')
>>> **class B(A): pass**
	>>> **class D(B, C): pass** # *Cautare implicita*
	>>> **x = D()** # *Poate depinde de tipul clasei*
	>>> **x.meth()** # *Ordinea clasica in v2.x*
	A.meth

4…

>>> **class D(B, C): meth = C.meth** *# <= Alegere explicita a metodei din C: stil nou (si v3.x)*

>>> **x = D()**

>>> **x.meth()**

C.meth

>>> **class D(B, C): meth = B.meth** *# <= Alegere explicita a metodei din B: stil clasic*

>>> **x = D()**

>>> **x.meth()**

A.meth

class D(B, C):

def meth(self): *# Redefinire mai jos*

...

C.meth(self) *# <= Alegere a metodei din C prin apelare*

- ***Alegerea explicita asigura portabilitatea codului!***

4...

- <u>Algoritmul MRO – Method Resolution Order</u>:
  1. Se listeaza toate clasele de la care o instanta mosteneste, folosindu-se DFLR
  2. Se elimina toate duplicatele cu exceptia ultimei aparitii din lista, pentru fiecare clasa
- <u>Vizualizarea MRO cu atributul (*tuple*)</u> ***class.__mro__***:

```
>>> class A: pass
>>> class B(A): pass # Romb=>ordine diferita in stilul nou
>>> class C(A): pass # Pe orizontala intai
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

```
>>> class A: pass
>>> class B(A): pass # Nonromb=>ordine clasica
>>> class C: pass # Adancime, apoi de la stanga la dreapta
>>> class D(B, C): pass
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>,
<class '__main__.C'>, <class 'object'>)
```

4...

- Exista si metoda ***class.mro()***:

```
>>> class X: pass
>>> class Y: pass
>>> class A(X): pass # Nonromb
>>> class B(Y): pass # Cu object se formeaza intotdeauna un romb!
>>> class D(A, B): pass
```

```
>>> D.mro()
[<class '__main__.D'>, <class '__main__.A'>, <class '__main__.X'>,
<class '__main__.B'>, <class '__main__.Y'>, <class 'object'>]
```

```
>>> D.mro() == list(D.__mro__)
True
```

```
>>> [cls.__name__ for cls in D.__mro__]
['D', 'A', 'X', 'B', 'Y', 'object']
```

- Exemplu, maparea atributelor per clase:

```
"""
Fisier mapattrs.py
mapattrs() mapeaza atributele per clasa din care provin. Cautarea este MRO sau DFLR.
"""
```

```
import pprint
def trace(X, label='', end='\n'):
    print(label + pprint.pformat(X) + end) # Afisare pretty, recursiva
```

4...

def *filterdictvals*(D, V):

"""

dict D cu intrarile cu valoarea V sterse:

filterdictvals(dict(a=1, b=2, c=1), 1) => {'b': 2}

"""

return {K: V2 for (K, V2) in D.items() if V2 != V}

def *invertdict*(D):

"""

dict D, inversat, cu valorile pe post de chei

invertdict(dict(a=1, b=2, c=1)) => {1: ['a', 'c'], 2: ['b']}

"""

def keysof(V):

return sorted(K for K in D.keys() if D[K] == V)

return {V: keysof(V) for V in set(D.values())}

def *dflr*(cls):

"""

Cautare clasica, DFLR, recursiva

"""

here = [cls]

for sup in cls.__bases__:

here += dflr(sup)

return here

def *inheritance*(instance):

"""

Mostenire MRO sau DFLR

"""

4…

```
    if hasattr(instance.__class__, '__mro__'):
        return (instance,) +
            instance.__class__.__mro__
    return [instance] + dflr(instance.__class__)
def mapattrs(instance, withobject=False,
    bysource=False):
    """
attr2obj este dict cu chei atributele si valori
    obiectul de la care sunt mostenite
withobject: False=fara clasa predefinita object
bysource: True=grupare per obiecte in loc de
    atribute
    """
    attr2obj = {}
    inherits = inheritance(instance)
    for attr in dir(instance):
        for obj in inherits:
            if hasattr(obj, '__dict__') and
    attr in obj.__dict__: # Slots, urmeaza...
                attr2obj[attr] = obj
                break
    if not withobject:
        attr2obj = filterdictvals(attr2obj,
    object)
    return attr2obj if not bysource else
    invertdict(attr2obj)
if __name__ == '__main__': # Autotestare
    print('Classic classes in 2.X, new-style in
    3.X')
    class A: attr1 = 1
    class B(A): attr2 = 2
    class C(A): attr1 = 3
```

4...

class D(B, C): pass

I = D()

print('Py=>%s' % I.attr1)

trace(inheritance(I), 'INH\n') *# Mostenire*

trace(mapattrs(I), 'ATTRS\n') *# Atr => Sursa*

trace(mapattrs(I, bysource=True), 'OBJS\n') *# Sursa => [Atr]*

print('New-style classes in 2.X and 3.X')

class A(object): attr1 = 1 *# "(object)" optional in v3.x*

class B(A): attr2 = 2

class C(A): attr1 = 3

class D(B, C): pass

I = D()

print('Py=>%s' % I.attr1)

trace(inheritance(I), 'INH\n')

trace(mapattrs(I), 'ATTRS\n')

trace(mapattrs(I, bysource=True), 'OBJS\n')

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

Classic classes in 2.X, new-style in 3.X

Py=>1

INH

[<__main__.D instance at 0x0000000002D81CC8>,

<class __main__.D at 0x0000000002D5AAC8>,

<class __main__.B at 0x0000000002D5A9A8>,

<class __main__.A at 0x0000000002D5A888>,

<class __main__.C at 0x0000000002D5AA68>,

4...

<class __main__.A at 0x000000002D5A888>]

ATTRS

{'__doc__': <class __main__.D at 0x000000002D5AAC8>,

'__module__': <class __main__.D at 0x000000002D5AAC8>,

'attr1': <class __main__.A at 0x000000002D5A888>,

'attr2': <class __main__.B at 0x000000002D5A9A8>}

OBJS

{<class __main__.A at 0x000000002D5A888>: ['attr1'],

<class __main__.B at 0x000000002D5A9A8>: ['attr2'],

<class __main__.D at 0x000000002D5AAC8>: ['__doc__', '__module__']}

New-style classes in 2.X and 3.X

Py=>3

INH

(<__main__.D object at 0x000000002D858C8>,

<class '__main__.D'>,

<class '__main__.B'>,

<class '__main__.C'>,

<class '__main__.A'>,

<type 'object'>)

ATTRS

{'__dict__': <class '__main__.A'>,

4…

'__doc__': <class '__main__.D'>,

'__module__': <class '__main__.D'>,

'__weakref__': <class '__main__.A'>,

'attr1': <class '__main__.C'>,

'attr2': <class '__main__.B'>}

OBJS

{<class '__main__.C'>: ['attr1'],

<class '__main__.D'>: ['__doc__', '__module__'],

<class '__main__.B'>: ['attr2'],

<class '__main__.A'>: ['__dict__', '__weakref__']}

Sumar

- Extinderea tipurilor predefinite
- Clase in stil nou
- ***Extensii ale claselor in stil nou***
- Metode statice si de clasa
- Decoratori si metaclase
- Decoratori de functii definiti de utilizator
- Decoratori de clase
- Metaclase
- Functia *super()*
- Proiectarea cu Clase

Sloturi

- **<u>Sloturi</u>** – folosesc un atribut de clasa numit __***slots***__
  - Limiteaza setul de atribute permise
  - Optimizeaza memoria – pentru multe instante, fara __*dict*__
  - Maresc viteza de executie(?)

```
>>> class limiter(object):
        __slots__ = ['age', 'name', 'job']

>>> x = limiter()
>>> x.age # Intai atribuire, apoi folosire!
AttributeError: age
```

```
>>> x.age = 40 # Ca date de instanta
>>> x.age
40
>>> x.ape = 1000 # Ilegal, nu este in __slots__
AttributeError: 'limiter' object has no attribute
    'ape'
```

  - Cu __*slots*__, instantele (implicit) nu au __*dict*__

```
>>> class C:
        __slots__ = ['a', 'b']
>>> X = C(); X.a = 1
```

```
>>> X.__dict__
AttributeError: 'C' object has no attribute
    '__dict__'
```

Sloturi...

- Atributele se pot accesa/modifica cu *getattr*, *setattr* si *dir*:

```
>>> getattr(X, 'a')
1
>>> setattr(X, 'b', 2)
>>> X.b
2
```

```
>>> 'a' in dir(X) # Si cu dir()
True
>>> 'b' in dir(X)
True
```

- Fara __*dict*__ nu se pot adauga atribute noi:

```
>>> class D:
        __slots__ = ['a', 'b']
        def __init__(self):
            self.d = 4 # Incorect
```

```
>>> X = D()
AttributeError: 'D' object has no attribute 'd'
```

- Dar ***'__dict__'*** se poate adauga in __*slots*__*!*

```
>>> class D:
        __slots__ = ['a', 'b', '__dict__']
        c = 3 # Atribut de clasa
        def __init__(self):
            self.d = 4 # Corect!
```

Sloturi...

```
>>> X = D()
>>> X.d
4
>>> X.c
3
>>> X.a # Intai asignare!
AttributeError: a
>>> X.a = 1
>>> X.b = 2
```

- Obiecte atat cu __*dict*__ cat si cu __*slots*__:

```
>>> X.__dict
{'d': 4}
>>> X.__slots__
['a', 'b', '__dict__']
>>> getattr(X, 'a'), getattr(X, 'c'), getattr(X, 'd') # getattr acceseaza sloturi, atribute de clasa si din dict-ul instantei
(1, 3, 4)
```

- Afisarea doar a atributelor de instanta (nu cu *dir*):

```
>>> for attr in list(X.__dict__) + X.__slots__: ... # Gresit...
        print(attr, '=>', getattr(X, attr), end=' ')
>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []): # Mai binisor...
d => 4 a => 1 b => 2 __dict__ => {'d': 4}
```

Sloturi...

- Mai multe __*slots*__ in superclase:

```
>>> class E:
        __slots__ = ['c', 'd'] # Cu __slots__
>>> class D(E):
        __slots__ = ['a', '__dict__'] # Si subclasa are __slots__
>>> X = D()
>>> X.a = 1; X.b = 2; X.c = 3 # Instanta vede reuniunea sloturilor!
>>> X.a, X.c
(1, 3)
```

```
>>> E.__slots__ # Dar __slots__ nu sunt concatenate!
['c', 'd']
>>> D.__slots__
['a', '__dict__']
>>> X.__slots__ # Instanta mosteneste cel mai de jos __slots__ !!
['a', '__dict__']
>>> X.__dict__ # Instanta are propriul __dict__
{'b': 2}
```

```
for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []): # Sloturile superclaselor lipsesc!
    print(attr, '=>', getattr(X, attr), end=' ')
b => 2 a => 1 __dict__ => {'b': 2}
```

```
>>> dir(X) # dir() vede toate atributele!
[..etc... 'a', 'b', 'c', 'd']
```

Sloturi...

- Accesul generic la atribute (virtuale):

```
>>> class Slotful:
        __slots__ = ['a', 'b', '__dict__']
        def __init__(self, data):
            self.c = data
>>> I = Slotful(3)
>>> I.a, I.b = 1, 2
>>> I.a, I.b, I.c # Acces normal la atribute
(1, 2, 3)
>>> I.__dict__ # Exista si __dict__
{'c': 3}
>>> [x for x in dir(I) if not x.startswith('__')]
['a', 'b', 'c']
>>> I.__dict__['c'] # __dict__ este sursa partiala
    de atribute
3
>>> getattr(I, 'c'), getattr(I, 'a') # dir+getattr
    cuprinde mai mult ca __dict__: sloturi,
    proprietati, descriptori, urmeaza...
(3, 1)
>>> for a in (x for x in dir(I) if not
    x.startswith('__')):
      print(a, getattr(I, a), end=' ')
a 1 b 2 c 3
```

Sloturi...

- Reguli de utilizare a sloturilor:
  1. Sloturi in subclase sunt inutile cand lipsesc din superclase – care vor contine __*dict*__ accesibil
  2. Sloturi in superclase sunt inutile cand lipsesc din subclase – iar apare __*dict*__...
  3. Redefinirea numelor din slotul unei clase ascunde slotul din superclasa
  4. Atributele de clasa inlocuiesc cele din slot – eroare
  5. Sloturile elimina __*dict*__ si nu permit alte atribute decat cele din slot, cu exceptia prezentei lui ***'__dict__'*** in slot.

```
>>> class C: pass # (1)
>>> class D(C): __slots__ = ['a']
>>> X = D()
>>> X.a = 1; X.b = 2
```

```
>>> X.__dict__
{'b': 2}
>>> D.__dict__.keys()
dict_keys(['__module__', '__slots__', 'a',
    '__doc__'])
```

Sloturi...

>>> **class C: __slots__ = ['a']** ***# (2)***
>>> **class D(C): pass**
>>> **X = D()**
>>> **X.a = 1; X.b = 2**
	>>> **X.__dict__**
	{'b': 2}
	>>> **C.__dict__.keys()**
	dict_keys(['__module__', **'__slots__'**, **'a'**, '__doc__'])

>>> **class C: __slots__ = ['a']** ***# (3)***	>>> **class D(C): __slots__ = ['a']**
>>> **class C: __slots__ = ['a']; a = 99** ***# (4)***	ValueError: 'a' in __slots__ conflicts with class variable

- Sloturile sunt efective daca sunt prezente in toate clasele si daca definesc doar nume noi fata de cele din superclase:

>>> **class C: __slots__ = ['a']** *# Nume diferite*
>>> **class D(C): __slots__ = ['b']**
>>> **X = D()**
>>> **X.a = 1; X.b = 2**
>>> **X.__dict__** *# Eliminat!*
	AttributeError: 'D' object has no attribute '__dict__'
	>>> **C.__dict__.keys(), D.__dict__.keys()**
	(dict_keys(['__module__', '__slots__', 'a', '__doc__']), dict_keys(['__module__', '__slots__', 'b', '__doc__']))

Sloturi...

- Viteza sloturilor:

```
# Fisier slots-test.py:
from __future__ import print_function
import timeit
base = """
Is = []
for i in range(1000):
    X = C()
    X.a = 1; X.b = 2; X.c = 3; X.d = 4
    t = X.a + X.b + X.c + X.d
    Is.append(X)
"""
stmt = """
class C: __slots__ = ['a', 'b', 'c', 'd']
""" + base
print('Slots =>', end=' ')
print(min(timeit.repeat(stmt, number=1000,
    repeat=3)))
stmt = """
class C: pass
""" + base
print('Nonslots=>', end=' ')
print(min(timeit.repeat(stmt, number=1000,
    repeat=3)))
```

```
C:\code>py -3 slots-test.py
Slots   => 0.4572967
Nonslots=> 0.5358057
```

```
C:\code>py -2 slots-test.py
Slots   => 0.5066377
Nonslots=> 0.5018881
```

Proprietati

- **<u>Proprietati</u>** – sunt modalitati de acces/asignare a atributelor
  - Sunt o alternativa pentru metodele inlocuite *__getattr__* si *__setattr__*
  - Se aseamana cu sloturile – atribute virtuale, fara __dict__, bazate pe descriptori (de clasa)
  - Se definesc cu functia predefinita ***property()*** sau cu sintaxa @
    - e.g. *name=property()* in corpul unei clase

```
>>> class operators: # Cu inlocuirea metodei __getattr__
      def __getattr__(self, name):
          if name == 'age':
              return 40
          raise AttributeError(name)
```

```
>>> x = operators()
>>> x.age # Se executa __getattr__ care intercepteaza atributele nedefinite
40
>>> x.name
AttributeError: name
```

Proprietati...

```
>>> class properties(object): # object in v2.x
        def getage(self):
                return 40
        age = property(getage, None, None, None) # (get, set, del, docs), sau cu @
```

```
>>> x = properties()
>>> x.age # Apel de getage
40
>>> x.name # Acces normal
AttributeError: 'properties' object has no attribute 'name'
```

- Proprietatile sunt mai usor de implementat:

```
>>> class properties(object):
        def getage(self):
                return 40
        def setage(self, value):
                print('set age: %s' % value)
                self._age = value
        age = property(getage, setage, None, None)
```

```
>>> x = properties()
>>> x.age # Apel de getage
40
>>> x.age = 42 # Apel de setage
set age: 42
>>> x._age # Fara apel de getage(), normal
42
```

Proprietati...

```
>>> x.age # getage()
40
>>> x.job = 'trainer' # Fara setage(), normal
>>> x.job # Fara getage(), normal
'trainer'
```

- Mai complicat cu inlocuire de operatori:

```
>>> class operators:
  def __getattr__(self, name): # La referinte nedefinite
    if name == 'age':
      return 40
    raise AttributeError(name)
  def __setattr__(self, name, value): # La toate asignarile
    print('set: %s %s' % (name, value))
    if name == 'age':
      self.__dict__['_age'] = value # Sau object.__setattr__()
    else:
      self.__dict__[name] = value
>>> x = operators()
>>> x.age # Apel de __getattr__
40
>>> x.age = 41 # Apel de  __setattr__
set: age 41
>>> x._age # Fara apel de __getattr__
41
>>> x.age # Apel de __getattr__
40
```

Proprietati/Alte Extensii

```
>>> x.job = 'trainer' # Apel de __setattr__
set: job trainer
>>> x.job # Fara apel de __getattr__, atr. definit
'trainer'
```

- Scriere cu sintaxa decoratorilor de functii, @:

```
class properties(object):
    @property # Urmeaza...
    def age(self): ...
    @age.setter
    def age(self, value): ...
```

- Metoda **__getattribute__**
  - Se aplica tuturor referintelor de atribute (si definite)
- **Descriptori**, cu metodele ***__get__***, ***__set__***, ***__delete__***
  - Intercepteaza citirea/scrierea/stergerea atributelor:

```
>>> class AgeDesc(object):
    def __get__(self, instance, owner): return 40
    def __set__(self, instance, value):
        instance._age = value
>>> class descriptors(object):
        age = AgeDesc()
```

Alte...

>>> **x = descriptors()**	>>> **x.age = 42** *# Apel de AgeDesc.__set__*
>>> **x.age** *# Apel de AgeDesc.__get__*	>>> **x._age** *# Fara apel de AgeDesc, normal*
40	42

- Alte extensii:
  - Metoda ***super()***
  - **Metaclase** – deriva din *type*, intercepteaza creatia claselor, furnizeaza comportament claselor

Sumar

- ❑ Extinderea tipurilor predefinite
- ❑ Clase in stil nou
- ❑ Extensii ale claselor in stil nou
- ❑ ***<u>Metode statice si de clasa</u>***
- ❑ Decoratori si metaclase
- ❑ Decoratori de functii definiti de utilizator
- ❑ Decoratori de clase
- ❑ Metaclase
- ❑ Functia *super()*
- ❑ Proiectarea cu Clase

Metode statice si de clasa

- Sunt metode apelabile fara o instanta:
  - Metode **statice**: functii fara instanta (*self*) dintr-o clasa
  - Metode **de clasa**: cu argument de tip clasa
- Necesita apelarea functiilor predefinite ***staticmethod*** si ***classmethod*** in clasa, sau cu sintaxa decoratorilor, @name
- In Python v3.x declaratia *staticmethod* nu este necesara in cazul metodelor fara instanta, apelabile doar prin numele clasei
- Rolul metodelor speciale:
  - Prelucrarea datelor asociate cu o clasa – metode statice
    - numarul de instante create, lista instantelor aflate in memorie
    - folosesc atribute de clasa, nu au argument *self*

Metode...

- Accesul la datele unei clase cu un argument de tip clasa – metode de clasa

- Metode statice in v2.x si v3.x
  - In v2.x metodele trebuie declarate statice spre a fi apelate fara o instanta, apelul fiind prin clasa sau instanta
  - In v3.x metodele nu trebuie declarate statice daca sunt apelate numai prin clasa, dar pentru apelul prin instanta declararea statica este necesara

```
class Spam: # Nu merge in v2.x, doar in v3.x daca calificare cu clasa, si nu prin instanta
    numInstances = 0
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances created: %s" % Spam.numInstances)
```

Metode...

>>> *# Python v2.7, nu merge:*
>>> **a = Spam()**
>>> **b = Spam()**
>>> **c = Spam()**
>>> **Spam.printNumInstances()** *# Via clasa, cere instanta ca prim argument!*
	TypeError: unbound method printNumInstances() must be called with Spam instance as first argument (got nothing instead)
	>>> **a.printNumInstances()** *# self transmis!*
	TypeError: printNumInstances() takes no arguments (1 given)

>>> *# Python v3.7*
>>> **a = Spam()**
>>> **b = Spam()**
>>> **c = Spam()**
	>>> **Spam.printNumInstances()** # *Merge via clasa*
	Number of instances created: 3
	>>> **a.printNumInstances()** *# self transmis!*
	TypeError: printNumInstances() takes 0 positional arguments but 1 was given

- Alternative (vechi) la metodele statice:
  - functie externa in acelasi modul cu clasa, care acceseaza atributul de clasa:

Metode...

def printNumInstances(): **print("Number of instances created: %s" % Spam.numInstances)**	**class Spam:** **numInstances = 0** **def __init__(self):** **Spam.numInstances += 1**

>>> **a, b, c = Spam(), Spam(), Spam()**
>>> **printNumInstances()** *# Cu functie*
Number of instances created: 3
	>>> **Spam.numInstances**
	3

- cu metoda normala, dar numarul de instante poate creste/eronat:

class Spam: **numInstances = 0** **def __init__(self): Spam.numInstances += 1**	**def printNumInstances(self):** **print("Number of instances created: %s" % Spam.numInstances)**

>>> **a, b, c = Spam(), Spam(), Spam()**
>>> **a.printNumInstances()** *# Cu metoda normala*
Number of instances created: 3
	>>> **Spam.printNumInstances(a)**
	Number of instances created: 3
	>>> **Spam().printNumInstances()** *# **Eroare!***
	Number of instances created: 4

Metode...

- Implementarea metodelor statice si de clasa:

```
# Fisier bothmethods.py:
class Methods:
    def imeth(self, x): # Metoda normala, cu self
        print([self, x])
    def smeth(x): # Metoda statica, fara self (instanta)
        print([x])
    def cmeth(cls, x): # Metoda de clasa, cu argument clasa, nu instanta
        print([cls, x])
    smeth = staticmethod(smeth) # smeth devine o metoda statica (sau cu @)
    cmeth = classmethod(cmeth) # cmeth devine o metoda de clasa (sau cu @)
```

```
>>> from bothmethods import Methods
>>> obj = Methods()
>>> obj.imeth(1) # Apel via instanta
[<bothmethods.Methods object at
    0x000001E73DA02D08>, 1]
>>> Methods.imeth(obj, 2) # Apel via clasa
[<bothmethods.Methods object at
    0x000001E73DA02D08>, 2]
```

```
>>> Methods.smeth(3) # Metoda statica, apel
    via clasa
[3]
>>> obj.smeth(4) # Metoda statica, apel via
    instanta
[4]
```

Metode...

```
>>> Methods.cmeth(5) # Metoda de clasa, devine apel de cmeth(Methods, 5)
[<class 'bothmethods.Methods'>, 5]
>>> obj.cmeth(6) # Metoda de clasa, apelata via instanta
[<class 'bothmethods.Methods'>, 6]
```

- <u>Numararea instantelor cu metode statice</u>:

```
class Spam:
    numInstances = 0 # Atribut de clasa, cu metoda statica
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances: %s" % Spam.numInstances)
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> a = Spam(); b = Spam(); c = Spam()
>>> Spam.printNumInstances() # Apel ca de functie simpla
Number of instances: 3
>>> a.printNumInstances() # Argument de tip instanta NU este transmis!
Number of instances: 3
```

- Metodele statice sunt locale clasei
- Sunt bine localizate
- Permit adaptarea prin mostenire:

Metode...

```
class Sub(Spam):
    def printNumInstances(): # Inlocuire a unei metode statice
        print("Extra stuff...")
        Spam.printNumInstances() # Apelul metodei din superclasa
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> a = Sub(); b = Sub()
>>> a.printNumInstances() # Apel din instanta
Extra stuff...
Number of instances: 2
>>> Sub.printNumInstances() # Din subclasa
Extra stuff...
Number of instances: 2
>>> Spam.printNumInstances() # Apelul versiunii originale
Number of instances: 2
```

```
>>> class Other(Spam): pass # Simpla mostenire a metodei statice (si a lui __init__)
>>> c = Other()
>>> Other.printNumInstances()
Number of instances: 3
>>> c.printNumInstances()
Number of instances: 3
```

Metode...

- Numararea instantelor cu metode de clasa:

```
class Spam:
    numInstances = 0 # Atribut de clasa, cu metoda de clasa
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s" % cls.numInstances)
    printNumInstances = classmethod(printNumInstances)
```

```
>>> a, b = Spam(), Spam()
>>> a.printNumInstances() # Primul argument va fi clasa lui a
Number of instances: 2
>>> Spam.printNumInstances() # Idem, primul argument este clasa
Number of instances: 2
```

  - Clasa – argument al metodei de clasa este cea mai specifica (jos, dreapta) :

```
>>> class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s %s" % (cls.numInstances, cls))
    printNumInstances = classmethod(printNumInstances)
```

Metode...

```
>>> class Sub(Spam):
        def printNumInstances(cls): # Inlocuire metoda de clasa
            print("Extra stuff...", cls)
            Spam.printNumInstances() # Dar si apelul metodei originale
        printNumInstances = classmethod(printNumInstances)
>>> x, y = Sub(), Spam()
>>> x.printNumInstances() # Apel din instanta subclasei
Extra stuff... <class '__main__.Sub'>
Number of instances: 2 <class '__main__.Spam'>
>>> Sub.printNumInstances() # Apel din subclasa
Extra stuff... <class '__main__.Sub'>
Number of instances: 2 <class '__main__.Spam'>
>>> y.printNumInstances() # Apel din instanta superclasei
Number of instances: 2 <class '__main__.Spam'>
>>> class Other(Spam): pass # Simpla mostenire
>>> z = Other()
>>> z.printNumInstances() # Argumentul implicit este clasa Other! (nu Spam)
Number of instances: 3 <class '__main__.Other'>
```

- **Observatie: o asignare a datelor va fi deci a clasei Other, nu Spam!**

Metode...

- Metode de clasa pentru numararea instantelor **fiecarei** clase:
  - Metodele statice + nume de clasa explicite: potrivite pentru prelucrarea datelor locale unei clase
  - Metodele de clasa: potrivite pentru date diferite din clasele unei ierarhii

```
>>> class Spam:
        numInstances = 0
        def count(cls): # Contor per clasa!
            cls.numInstances += 1 # cls este clasa imediat deasupra instantei
        def __init__(self):
            self.count() # self.__class__ este transmis lui count()
        count = classmethod(count)
>>> class Sub(Spam):
        numInstances = 0
        def __init__(self): # Inlocuire __init__
            Spam.__init__(self)
>>> class Other(Spam): # Mostenire __init__
        numInstances = 0
>>> x = Spam(); y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances # Date per clasa!
(1, 2, 3)
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```

Sumar

- Extinderea tipurilor predefinite
- Clase in stil nou
- Extensii ale claselor in stil nou
- Metode statice si de clasa
- ***<u>Decoratori si metaclase</u>***
- Decoratori de functii definiti de utilizator
- Decoratori de clase
- Metaclase
- Functia *super()*
- Proiectarea cu Clase

Decoratori si metaclase

- <u>Decoratorii</u> permit adaugarea de cod in functii sau clase
- Decoratorii de **functii** – extind definitia functiilor si metodelor prin includere intr-o alta functie – *metafunctie*
  - Pot extinde functiile cu cod suplimentar (e.g. pentru logare apeluri)
  - La apel, joaca rolul unei delegatii, pentru o functie sau metoda (nu intreaga interfata a obiectului).
  - Python ofera decoratori predefiniti: metode statice, de clasa si *property*
- Decoratorii de **clase** – extind definitia claselor, la nivel de obiect si interfata; au rol asemanator cu *metaclasele* (dar mai simpli)
  - Pot afecta definitia claselor, iar a instantelor – cu tehnica delegarii
  - Sunt mai simpli decat metaclasele

Decoratori de functii

- Sintaxa: **@metafunctie** pe randul dinaintea unui ***def***

```
class C:
    @staticmethod # Decorator
    def meth():
        ...
```

```
class C:
    def meth():
        ...
    meth = staticmethod(meth) # Cod echivalent de redefinire a lui meth
```

- Decoratorul (*staticmethod*) poate executa cod suplimentar la fiecare apel
- Poate sa returneze rezultatul functiei originale sau un obiect (*proxy*) ce intai executa codul suplimentar si apoi apeleaza functia originala

Decoratori...

- Exemplu – cu metoda statica:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances += 1
    @staticmethod
    def printNumInstances():
        print("Number of instances created: %s" % Spam.numInstances)
```

```
>>> from spam_static_deco import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances() # Apel din clasa
Number of instances created: 3
>>> a.printNumInstances() # Apel din instanta
Number of instances created: 3
```

- Exemplu – si cu *classmethod* si *property*:

```
class Methods(object): # object pt. v2.x (property)
    def imeth(self, x): # Metoda normala, de instanta, cu self
        print([self, x])
    @staticmethod
    def smeth(x): # Metoda statica, fara self
        print([x])
```

Decoratori...

```
@classmethod
def cmeth(cls, x): # Metoda de clasa, cu argument clasa, nu instanta
    print([cls, x])
@property # Proprietate, atributul name
def name(self):
    return 'Bob ' + self.__class__.__name__
>>> from bothmethods_decorators import Methods
```

```
>>> obj = Methods()
>>> obj.imeth(1)
[<__main__.Methods object at 0x000002038F8042C8>, 1]
>>> obj.smeth(2)
[2]
>>> obj.cmeth(3)
[<class '__main__.Methods'>, 3]
>>> obj.name
'Bob Methods'
```

Sumar

- Extinderea tipurilor predefinite
- Clase in stil nou
- Extensii ale claselor in stil nou
- Metode statice si de clasa
- Decoratori si metaclase
- ***Decoratori de functii definiti de utilizator***
- Decoratori de clase
- Metaclase
- Functia *super()*
- Proiectarea cu Clase

Decoratori de functii definiti de utilizator

- Exemplu: cu operatorul __**call**__

```
class tracer:
    def __init__(self, func):
        self.calls = 0 # Contor de apeluri
        self.func = func # Functia originala
    def __call__(self, *args): # La apel: cod adaugat plus apelul functiei originale
        self.calls += 1
        print('call %s to %s' % (self.calls,
              self.func.__name__))
        return self.func(*args)

@tracer # Efect: spam = tracer(spam)
def spam(a, b, c): # spam() este inclusa intr-un decorator! Devine instanta a clasei tracer.
    return a + b + c
```

```
>>> print(spam(1, 2, 3)) # De fapt, apel al decoratorului
call 1 to spam
6
```

```
>>> print(spam('a', 'b', 'c')) # Se executa __call__ din clasa tracer
call 2 to spam
abc
```

- Limitari: decorator doar pentru functii cu argumente pozitionale, si nu pentru metode de clasa

Decoratori...

- Exemplu: cu o functie decorator

```
def tracer(func): # Retine functia originala
    def oncall(*args): # La apeluri ulterioare
        oncall.calls += 1 # Atribut de functie
        print('call %s to %s' % (oncall.calls, func.__name__))
        return func(*args)
    oncall.calls = 0 # Initializarea atributului de functie
    return oncall
```

```
class C:
    @tracer # Efect: spam = tracer(spam) care returneaza functia oncall()
    def spam(self,a, b, c): return a + b + c
```

```
>>> x = C()
>>> print(x.spam(1, 2, 3)) # De fapt, apel al decoratorului, aici oncall()
call 1 to spam
6
```

```
>>> print(x.spam('a', 'b', 'c')) # Se executa oncall()
call 2 to spam
abc
```

Sumar

- ❑ Extinderea tipurilor predefinite
- ❑ Clase in stil nou
- ❑ Extensii ale claselor in stil nou
- ❑ Metode statice si de clasa
- ❑ Decoratori si metaclase
- ❑ Decoratori de functii definiti de utilizator
- ❑ ***Decoratori de clase***
- ❑ Metaclase
- ❑ Functia *super()*
- ❑ Proiectarea cu Clase

Decoratori de clase

- Sintaxa:

```
def decorator(aClass): ...

@decorator # Decorator de clasa
class C: ...
```

```
def decorator(aClass): ...

class C: ... # Cod echivalent
C = decorator(C)
```

- Exemplu: numararea instantelor per clasa

```
def count(aClass):
    aClass.numInstances = 0
    return aClass # Returneaza chiar clasa, nu proxy

@count
class Spam: ... # Efect: Spam = count(Spam)
```

```
@count
class Sub(Spam): ... # numInstances = 0 nenecesar

@count
class Other(Spam): ...
```

```
@count # Merge si cu simple functii!
def spam(): pass # Efect: spam = count(spam)
# Atributul spam.numInstances este zero
```

Decoratori...

- <u>Exemplu</u>: cu un proxy

```
def decorator(cls):
    class Proxy:
        def __init__(self, *args): # Se creeaza o instanta a clasei cls
            self.wrapped = cls(*args)
        def __getattr__(self, name): # La citirea atributelor
            return getattr(self.wrapped, name) # Atributul instantei incluse
    return Proxy
```

```
@decorator
class C: ... # Efect: C = decorator(C)

X = C() # Instanta a clasei Proxy care include o instanta a clasei C – care returneaza X.attr

>>> type(X)
<class '__main__.decorator.<locals>.Proxy'>
```

Sumar

- Extinderea tipurilor predefinite
- Clase in stil nou
- Extensii ale claselor in stil nou
- Metode statice si de clasa
- Decoratori si metaclase
- Decoratori de functii definiti de utilizator
- Decoratori de clase
- ***Metaclase***
- Functia *super()*
- Proiectarea cu Clase

Metaclase

- Metaclasele deriva din clasa *type* si controleaza creatia unui obiect nou prin redefinirea metodelor __*new*__ sau __*init*__ ale clasei *type.* Sintaxa:

class Meta(type):

def __new__(meta, classname, supers, classdict):

***...cod suplimentar + creatie de clasa prin apel de type*...**

class C(metaclass=Meta):

...clasa creata via Meta... # Efect C = Meta('C', (), {...})

- **Detalii, urmeaza...**

Sumar

- ❑ Extinderea tipurilor predefinite
- ❑ Clase in stil nou
- ❑ Extensii ale claselor in stil nou
- ❑ Metode statice si de clasa
- ❑ Decoratori si metaclase
- ❑ Decoratori de functii definiti de utilizator
- ❑ Decoratori de clase
- ❑ Metaclase
- ❑ **Functia *super()***
- ❑ Proiectarea cu Clase

Functia predefinita *super()*

- Poate fi folosita la apelarea metodelor din superclase
- Modelul traditional – numirea explicita a superclaselor:

```
>>> class C: # In Python v2.X si v3.X
        def act(self):
            print('spam')
>>> class D(C):
        def act(self):
            C.act(self) # Calificare cu superclasa, argument self
            print('eggs')
>>> X = D()
>>> X.act()
spam
eggs
```

- Cu *super* si mostenire simpla:

```
>>> class C: # In Python v3.X
        def act(self):
            print('spam')
>>> class D(C):
        def act(self):  # Superclasa generica, fara argument self
            super().act()
            print('eggs')
>>> X = D(); X.act()
spam
eggs
```

Functia...

- *super* poate fi apelata numai din metode:

```
>>> super # Un obiect proxy...
<class 'super'>
>>> super()
SystemError: super(): no arguments
>>> class E(C):
        def method(self): # self e implicit in super
                proxy = super() # Corect, doar intr-o metoda
                print(proxy) # Afisarea obiectului proxy
                proxy.act() # Fara self: apel al metodei act din superclasa
>>> E().method()
<super: <class 'E'>, <E object>>
spam
```

Functia...

- *super* si mostenirea multipla:

```
>>> class A: # In Python v3.X
        def act(self): print('A')
>>> class B:
        def act(self): print('B')
>>> class C(A):
        def act(self):
            super().act() # super cu mostenire simpla este OK
>>> X = C()
>>> X.act()
A
>>> class C(A, B): # Mostenire multipla, aceeasi metoda act
        def act(self):
            super().act() # Se executa o singura metoda, cea aflata in clasa din stanga
>>> X = C()
>>> X.act()
A
>>> class C(B, A):
        def act(self):
            super().act() # A.act() nu se mai executa...
>>> X = C()
>>> X.act()
B
```

Functia...

- In loc de *super()*, mai bine cu selectie explicita a superclaselor:

```
>>> class C(A, B): # Model traditional
        def act(self):
            A.act(self) # Corect si in mostenire simpla si si multipla
            B.act(self) # Corect in Python v3.X si v2.X
>>> X = C() # super() este inutil...
>>> X.act()
A
B
```

- *super()* si operatori inlocuiti (__X__):

```
>>> class C: # In Python v3.X
        def __getitem__(self, ix): # Indexare
            print('C index')
>>> class D(C):
        def __getitem__(self, ix): # Redefinire pentru extindere
            print('D index')
            C.__getitem__(self, ix) # Apel explicit, traditional, OK
            super().__getitem__(ix) # Apel cu super si nume, OK
            super()[ix] # Expresie cu operator, NU (__getattribute__)
```

Functia...

```
>>> X = C(); X[99]
C index
>>> X = D(); X[99]
D index
C index
C index
TypeError: 'super' object is not subscriptable
```

- *super()* in Python v2.x:

```
>>> class C(object): # In Python 2.X: cu clase in
      stil nou, derivate din object
    def act(self):
        print('spam')
>>> class D(C):
    def act(self):
        super(D, self).act() # v2.X: apel diferit
        print('eggs')
>>> X = D()
>>> X.act()
spam
eggs
```

Functia...

- Avantajele folosirii lui *super()*:
  - Modificarea arborilor de mostenire in timpul executiei:

>>> **class X:**
def m(self): print('X.m')
>>> **class Y:**
def m(self): print('Y.m')
>>> **class C(X):** *# Mostenire a lui X*
def m(self): super().m() *# Cu super()*
	>>> **i = C()**
	>>> **i.m()**
	X.m
	>>> **C.__bases__ = (Y,)** *# <u>Modificare a superclasei la executie</u>*
	>>> **i.m()**
	Y.m

>>> **class C(X):**
def m(self): C.__bases__[0].m(self) *# <u>Cod special, fara super()</u>*
>>> **i = C()**
>>> **i.m()**
	X.m
	>>> **C.__bases__ = (Y,)** *# Acelasi efect*
	>>> **i.m()**
	Y.m

Functia...

- Apelul metodelor cu mostenire multipla – *super* folosit peste tot!
  - aplicabilitate in cazul mostenirii multiple cu forma de romb
  - metoda cu acelasi nume, e.g. constructorul, __*init*__, este apelata in ordinea MRO, fiecare clasa fiind vizitata o singura data

```
>>> class A:
        def __init__(self): print('A.__init__')
>>> class B(A):
        def __init__(self): print('B.__init__'); A.__init__(self)
>>> class C(A):
        def __init__(self): print('C.__init__'); A.__init__(self)
>>> class D(B, C):
        def __init__(self): # Model traditional
            B.__init__(self) # Ambele superclase
            C.__init__(self)
>>> x = D() # Metoda __init__ a clasei A este apelata de doua ori!!
B.__init__
A.__init__
C.__init__
A.__init__
```

Functia...

```
>>> class A:
        def __init__(self): print('A.__init__')
>>> class B(A):
        def __init__(self):
            print('B.__init__')
            super().__init__()
>>> class C(A):
        def __init__(self):
            print('C.__init__')
            super().__init__()
>>> class D(B, C): pass
```

```
>>> x = D() # In ordinea MRO a clasei D, sunt vizitate pe rand toate clasele din arbore
B.__init__
C.__init__
A.__init__
>>> B.__mro__
(<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

Functia...

- Limitare, in afara de clasa *object*, o alta clasa poate fi necesara in rolul lui *object* (incheie rombul, ancorare)
  - Metodele apelate de *super* trebuie sa existe, sa aiba aceeasi semnatura (argumente) si toate in afara de ultima (e.g. *object*) trebuie sa foloseasca *super:*

```
>>> class B:
        def __init__(self):
            print('B.__init__')
            super().__init__()
>>> class C:
        def __init__(self):
            print('C.__init__')
            super().__init__()
```

```
>>> x = B() # object este implicit ultimul in MRO
B.__init__
>>> x = C()
C.__init__
>>> class D(B, C): pass
>>> x = D() # In ordinea din MRO al lui D
B.__init__
C.__init__
```

Functia...

- Limitare, toate clasele trebuie sa foloseasca *super*:

```
>>> class C:
        def __init__(self):
            print('C.__init__')
            super().__init__()
>>> class B: # Clasa fara super!
        def __init__(self): print('B.__init__')
>>> class D(B, C):
        def __init__(self):
            print('D.__init__')
            super().__init__()
```

```
>>> X = D() # Lipseste apelul lui C.__init__
D.__init__
B.__init__
```

Functia...

- Limitare, ordinea apelurilor este impusa de MRO:

```
>>> class B:
        def __init__(self):
                print('B.__init__')
                super().__init__()
>>> class C:
        def __init__(self):
                print('C.__init__')
                super().__init__()
>>> class D(B, C): # Ordinea necesara difera de MRO!
        def __init__(self):
                print('D.__init__')
                C.__init__(self)
                B.__init__(self)
```

```
>>> X = D() # Dupa B urmeaza C – ordinea MRO apeleaza metoda __init__ a lui C de doua ori
D.__init__
C.__init__
B.__init__
C.__init__
>>> [c.__name__ for c in D.__mro__]
['D', 'B', 'C', 'object']
```

Functia...

- Redefinirea metodelor:

```
>>> class A:
        def method(self): print('A.method'); super().method()
>>> class B(A): # Mostenire
        def method(self): print('B.method'); super().method()
>>> class C:
        def method(self): print('C.method') # Fara super, ancorare
>>> class D(B, C):
        def method(self): print('D.method'); super().method()
>>> X = D() # Apel in ordinea MRO
>>> X.method()
D.method
B.method
A.method
C.method
```

```
>>> class B(A):
        def method(self): print('B.method') # Redefinire
>>> class D(B, C):
        def method(self): print('D.method'); super().method()
>>> X = D() # Lipseste C.method!!
>>> X.method()
D.method
B.method
```

Functia...

- Clase cu mostenire mixta
  - *super* selecteaza urmatoarea clasa din MRO **care implementeaza** metoda ceruta:

```
>>> class A:
        def other(self): print('A.other')
>>> class Mixin(A):
        def other(self): print('Mixin.other'); super().other()
>>> class B:
        def method(self): print('B.method')
>>> class C(Mixin, B):
        def method(self): print('C.method'); super().other(); super().method()
>>> C().method() # Desi Mixin nu implementeaza method, ea este gasita mai tarziu in MRO, in B
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.Mixin'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>)
```

Functia...

- Idem, chiar daca una din ramuri nu foloseste *super*:

```
>>> class C(B, Mixin):
        def method(self): print('C.method'); super().other(); super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.Mixin'>,
<class '__main__.A'>, <class 'object'>)
```

- Idem, chiar cu mostenire in forma de romb:

```
>>> class A:
        def other(self): print('A.other')
>>> class Mixin(A):
        def other(self): print('Mixin.other'); super().other()
>>> class B(A):
        def method(self): print('B.method')
>>> class C(Mixin, B):
        def method(self): print('C.method'); super().other(); super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
```

Functia...

- Idem, si in ordine diferita:

```
>>> class C(B, Mixin):
      def method(self): print('C.method'); super().other(); super().method()
>>> C().method()
C.method
Mixin.other
A.other
B.method
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.Mixin'>,
<class '__main__.A'>, <class 'object'>)
```

- Problema, daca metode cu **acelasi** nume sunt pe ramuri diferite:

```
>>> class A:
      def method(self): print('A.method')
>>> class Mixin(A):
      def method(self): print('Mixin.method'); super().method()
>>> Mixin().method()
Mixin.method
A.method
>>> class B(A):
      def method(self): print('B.method') # Cu super  s-ar apela A dupa B
>>> class C(Mixin, B):
      def method(self): print('C.method'); super().method()
```

Functia...

```
>>> C().method() # A.method() lipseste!
C.method
Mixin.method
B.method
```

- Apelurile directe rezolva problema:

```
>>> class A:
        def method(self): print('A.method')
>>> class Mixin(A):
        def method(self): print('Mixin.method'); A.method(self) # C este irelevant
>>> class C(Mixin, B):
        def method(self): print('C.method'); Mixin.method(self)
>>> C().method()
C.method
Mixin.method
A.method
```

Functia...

- Limitare – metodele apelate cu *super* trebuie sa aiba aceeasi semnatura/argumente:

```
>>> class Employee:
        def __init__(self, name, salary): # Superclasa comuna
            self.name = name
            self.salary = salary
>>> class Chef(Employee):
        def __init__(self, name):
            super().__init__(name, 50000) # Cu super
>>> class Server(Employee):
        def __init__(self, name):
            super().__init__(name, 40000) # Cu super
```

```
>>> class TwoJobs(Chef, Server): pass
>>> tom = TwoJobs('Tom')
TypeError: __init__() takes 2 positional arguments but 3 were given
>>> TwoJobs.__mro__
(<class '__main__.TwoJobs'>, <class '__main__.Chef'>, <class '__main__.Server'>, <class '__main__.Employee'>, <class 'object'>)
```

- **Server.__init__, cu doua argumente, este apelat cu trei argumente de apelul super().__init__(name, 50000) din Chef!**

Functia...

- In concluzie, metoda apelata de *super* trebuie:
  - sa existe
  - sa aiba aceeasi semnatura (problema la constructori – __init__)
  - sa foloseasca *super* (cu exceptia ultimei)

- Dezavantaje *super*:
  - Difera intre v2.x si v3.x
  - In v3.x are limitari cu operatori (__X__) si mostenirea multipla
  - In v2.x codul este prea complex
- Avantaje *super*:
  - Rezolva apelul metodelor cu acelasi nume in arbori de mostenire multipla (cu folosire peste tot!)

Sumar

- Extinderea tipurilor predefinite
- Clase in stil nou
- Extensii ale claselor in stil nou
- Metode statice si de clasa
- Decoratori si metaclase
- Decoratori de functii definiti de utilizator
- Decoratori de clase
- Metaclase
- Functia *super()*
- ***Proiectarea cu Clase***

Proiectarea cu Clase

- Modificarea atributelor de clasa poate avea efecte laterale:

```
>>> class X:
        a = 1 # Atribut de clasa
>>> I = X()
>>> I.a # Mostenit de instanta
1
>>> X.a
1
>>> X.a = 2 # Nu numai X este modificat!
>>> I.a # Si I este modificat
2
>>> J = X() # J mosteneste schimbarile executate deja
>>> J.a
2
```

- Efect de inregistrare (*struct* in C):

```
class Record: pass
X = Record()
X.name = 'bob'
X.job = 'Pizza maker'
```

Proiectarea...

- Modificarea atributelor de clasa <u>modificabile *in-place*</u> poate avea efecte laterale:

```
>>> class C:
        shared = [] # Atribut de clasa modificabil
        def __init__(self):
            self.perobj = [] # Atribut de instanta
>>> x = C() # Doua instante
>>> y = C() # Ambele partajeaza atributul shared
>>> y.shared, y.perobj
([], [])
>>> x.shared.append('spam') # y va vedea
    modificarea
>>> x.perobj.append('spam') # Impact
    local/instanta
>>> x.shared, x.perobj
(['spam'], ['spam'])
>>> y.shared, y.perobj # y vede schimbarile
    facute de x
(['spam'], [])
>>> C.shared # Plasat in clasa!
['spam']
```

Proiectarea...

- Ordinea mostenirii multiple conteaza
  - De la stanga la dreapta in lista superclaselor
  - Cu asignare manuala:

```
class ListTree:
    def __str__(self): ...
    def other(self): ...
class Super:
    def __str__(self): ...
    def other(self): ...
class Sub(ListTree, Super): # __str__ este din ListTree – care este primul, in stanga
    other = Super.other # other este ales de la Super
    def __init__(self):
...
x = Sub() # Sub este cautata prima, apoi ListTree/Super
```

Proiectarea...

- Domeniul de valabilitate in metode si clase:

```
def generate():
    class Spam: # Spam este nume local in functia generate
        count = 1
        def method(self):
            print(Spam.count) # Spam este vizibil, din domeniul functiei generate(), cu regula E din LEGB
    return Spam()
generate().method()
```

  - Metodele (*def*) nu pot vedea spatiul de nume al clasei, decat prin calificare: Spam.count sau self.count !
  - Incluziunea se poate evita:

```
def generate():
    return Spam() # Spam este vazut cu G din LEGB
class Spam: # Definitie in modul
    count = 1
    def method(self):
        print(Spam.count) # Spam este in G
generate().method()
```

Proiectarea...

- Totusi, incluziunea este utila la fabrici (de clase):

```
>>> def generate(label): # Returneaza o clasa
        class Spam:
            count = 1
            def method(self):
                print("%s=%s" % (label, Spam.count))
        return Spam
```

```
>>> aclass = generate('Gotchas')
>>> I = aclass()
>>> I.method()
Gotchas=1
```

- Alte probleme in proiectarea claselor:
  - Alegerea atributelor – de clasa sau de instanta
  - Apelarea constructorilor (__*init*__) din superclase
  - Operatorii predefiniti, cu __*getattr*__, trebuie redefiniti in subclasa
- KISS – ierarhii scurte si flat!