Programarea calculatoarelor si limbaje de programare II

Sintaxa in detaliu a claselor Inlocuirea operatorilor

Universitatea Politehnica din Bucureşti

Sumar

- ***<u>Sintaxa claselor</u>***
- Inlocuirea operatorilor

Sintaxa instructiunii *class*

- ***class*** este o instructiune executabila:
  - nu o declaratie ca in C++
  - ca *def:* este executabila si creeaza un obiect de tip clasa si ii asigneaza un nume
  - de obicei, class se executa la importarea modulului ce o cuprinde
- ***class*** este o instructiune compusa:

```
class name(superclass,...): # Antet, name asignat; superclase intre paranteze
    # Corpul clasei (indentat)
    attr = value              # Date partajate la nivel de clasa
    def method(self,...):     # Metode ale clasei. e.g. __init__, constructor
        self.attr = value     # Date ale unei instante a clasei
```

Sintaxa...

- ***class*** creeaza un spatiu de nume populat cu atribute ale clasei, rezultate din asignari efectuate la executia corpului clasei:
  - ca *def*, asignarile din clasa creeaza un spatiu local
  - ca modulele, asignarile produc atribute, accesibile numai prin calificare (de clasa sau de instanta)
- Spatiile de nume ale claselor servesc la rezolvarea mecanismului de mostenire a atributelor – daca lipsesc din instanta sau clasa atunci sunt cautate in superclase
- Orice instructiune poate fi inclusa in corpul unei clase:

```
>>> class SharedData:
        spam = 42 # Atribut de clasa
>>> x = SharedData() # Instanta x
>>> y = SharedData() # Instanta y
>>> x.spam, y.spam # Mostenire din clasa
(42, 42)
```

Sintaxa...

- Atributul din clasa se acceseaza prin calificare cu numele clasei:

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
```

(99, 99, 99)

- Asignarea atributelor de instanta creeaza/modifica numai instanta (nu si clasa):

```
class MixedNames: # Instructiunea class
    data = 'spam' # data, atribut al clasei
    def __init__(self, value): # Constructor
        self.data = value # Atribut al instantei
    def display(self):
        print(self.data, MixedNames.data) # Afisare atribut de instanta si de clasa, cu respectivele calificari
```

```
>>> x = MixedNames(1) # x, instanta
>>> y = MixedNames(2) # y, alta instanta
>>> x.display(); y.display() # Atribute de instanta diferite, acelasi atribut de clasa
```

1 <u>spam</u>
2 <u>spam</u>

Metode – sintaxa

- Metodele clasei sunt instructiunile *def* din corpul clasei
  - confera comportament clasei
  - se executa ca orice functie, dar primul argument este o instanta, **self**

Apelul ***class.method(instance, args...)*** este echivalent cu:

instance.method(args...)

- Metoda este identificata prin mostenire, ca apartinand clasei sau unei superclase
- Denumirea ***self*** este conventionala – pozitia conteaza, primul argument
  - *self* este folosit la calificarea tuturor atributelor de instanta

Metode...

- Exemple:

class NextClass: *# Antet clasa + corp*
def printer(self, text): *# Antet metoda + corp de metoda; <u>doua argumente, primul este self</u>*
self.message = text *# Creare atribut de instanta*
print(self.message) *# Acces la atributul de instanta – ambele folosesc calificari cu self*
	>>> **x = NextClass()** *# "Apel" de clasa, rezulta instanta*
	>>> **x.printer('instance call')** *# Apel de metoda a instantei – mostenita de la clasa, <u>un singur argument</u>, self = x este implicit*
	instance call
	>>> **x.message** *# Atributul instantei, modificat*
	'instance call'

<u># Apel echivalent, ca metoda a clasei:</u> >>> **NextClass.printer(x, 'class call')** *# Apel direct de metoda a clasei, cu **x** ca prim argument* class call	>>> **x.message** *# Atributul instantei este din nou modificat* 'class call'
<u># Eroare, daca instanta nu apare ca prim argument:</u> >>> **NextClass.printer('bad call')**	TypeError: printer() missing 1 required positional argument: 'text'

Metode...

- Apelul direct al constructorului din superclasa
  - Garanteaza executia metodei __**init**__ a superclasei
    - __*init*__, ca orice atribut, este gasit doar o data, prin mostenire

```
class Super:                                    I = Sub(1, 2)
    def __init__(self, x):
        ...constructie...
class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x) # Apelul
    explicit al metodei __init__ a superclasei
        ...cod specific... # Alte initializari
```

- Alte tipuri de metode – in cursurile urmatoare:
  - metode statice (fara self)
  - metode de clasa (argument clasa)

Mostenirea atributelor

- Se produce in expresii de tipul: ***obiect.atribut***, unde obiectul este o instanta sau o clasa
- Arborele de cautare a atributelor este format din:
  - Atributele instantei se genereaza prin calificarea cu *self* in metodele clasei
  - Atributele clasei sunt create cu asignari in instructiunea *class*
  - Legatura cu superclasele se face prin listarea claselor intre paranteze rotunde in antetul instructiunii *class*
- Atributele se cauta in arbore incepand cu instanta, apoi clasa din care a fost generata instanta, apoi in toate superclasele clasei (de la stanga la dreapta)

Mostenirea...

Objects

Program

Superclass

Superclass

Class

Instance

object.attr?

```
class S1:

class S2:
```

```
class X(S1, S2):
    def attr(self,...):
        self.attr = V
```

```
object = X()
```

- Fiecare calificare a instantei produce o alta cautare in arbore

Mostenirea...

- Specializarea metodelor mostenite
  - Este permisa de algoritmul de cautare – atributele mai jos in arbore sunt gasite primele
  - Tehnici de redefinire a metodelor:
    - inlocuire completa
    - extindere, cu apel a metodei din superclasa
    - furnizare de atribute pentru superclasa

```
>>> class Super:
        def method(self):
            print('in Super.method')
>>> class Sub(Super): # Mosteneste pe Super
        def method(self): # Inlocuire metoda
            print('starting Sub.method')
            Super.method(self) # Apelul metodei din superclasa
            print('ending Sub.method')
            ...Extindere cod...
```

Mostenirea...

```
>>> x = Super() # x, instanta a lui Super
>>> x.method() # Apel de Super.method(x)
in Super.method

>>> x = Sub() # x, instanta a lui Sub
>>> x.method() # Apel de Sub.method(x), care
    apeleaza Super.method(x)
starting Sub.method
in Super.method
ending Sub.method
```

- Clasificarea tehnicilor de interfatare a claselor:
  - **Super**: cu o metoda care deleaga actiunea intr-o subclasa
  - **Mostenitor**: mosteneste totul de la Super
  - **Inlocuitor**: redefineste o metoda a lui Super
  - **Extensor**: specializeaza o metoda a lui Super pe care o si apeleaza
  - **Furnizor**: implementeaza actiunea din metoda *delegate* a lui Super

Mostenirea...

- Exemplu, fisierul *specialize.py*:

```
class Super:
    def method(self):
        print('in Super.method')
    def delegate(self):
        self.action() # ? din subclasa!
class Inheritor(Super): pass # Mostenitor
class Replacer(Super): # Inlocuitor
    def method(self):
        print('in Replacer.method')
class Extender(Super): # Extensor
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')
class Provider(Super): # Furnizor al lui action
    def action(self):
        print('in Provider.action')

if __name__ == '__main__': # Autotestare
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '... ') # Fiecare clasa are atributul __name__
        klass().method() # instanta klass() apeleaza metoda method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

Mostenirea...

- Output ***py -3 specialize.py***:

```
Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

- **Superclase abstracte**:
  1. Apelul *x.delegate()* gaseste metoda *delegate* in superclasa
  2. In metoda *Super.delegate*, calificarea *self.action()* va genera o noua cautare – metoda *action* este gasita in subclasa *Provider* fiindca *self* este o instanta a acestei clase

Mostenirea...

- *Super* este o clasa **abstracta** – care conteaza pe subclase pentru furnizarea implementarilor lipsa
- Clasele abstracte pot pune in evidenta metodele abstracte cu instructiunea ***assert*** sau provocand o exceptie de tipul ***NotImplementedError***:

```
class Super: # Cu assert
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!' # Daca apelata...
```

```
>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
# assert <expresie>, <argumente pentru exceptia AssertionError daca expresie este False>
```

Mostenirea...

class Super: *# <u>Cu raise</u>*
def delegate(self):
self.action()
def action(self):
raise NotImplementedError('action must be defined!')
	>>> **X = Super()**
	>>> **X.delegate()**
	NotImplementedError: action must be defined!

- Daca *action* lipseste si in subclasa, iar eroare:

>>> **class Sub(Super): pass** >>> **X = Sub()**	>>> **X.delegate()** NotImplementedError: action must be defined!

>>> **class Sub(Super):** *# <u>Corect</u>*
def action(self): print('spam')
	>>> **X = Sub()**
	>>> **X.delegate()**
	spam

- <u>Impiedicarea</u> instantierii unei clase abstracte se face cu *metaclass* si cu decoratorul *@abstractmethod*

Spatii de nume

- Reguli pentru gasirea numelor din spatii de nume:
  - Nume simple, necalificate, **X**, sunt cautate in domenii de valabilitate a variabilelor
  - Numele calificate, **obiect.atribut**, sunt gasite in spatiile de nume ale obiectelor
  - In domenii de valabilitate se initializeaza spatiile de nume ale obiectelor (module si clase)
- In cazul *obiect.atribut*, intai este cautat obiectul, in domeniile de valabilitate, apoi atributul lui in spatiile de nume ale obiectelor

Spatii...

- **Numele simple** – globale daca nu au fost asignate local
  - Asignarea – **X = valoare**
    - produce/creeaza nume locale, cu exceptia declararii cu *global* sau *nonlocal* (in v3.x)
  - Referirea – **X**
    - cauta pe X cu regula conturului LEGB. Clasele **nu** sunt cautate!
  - Colectii iterative, clauza *try*: adauga un spatiu local suplimentar
- **Numele calificate**:
  - Asignarea – **obiect.X = expresie**
    - creeaza/modifica doar atributul X al obiectului
  - Referirea – **obiect.X**
    - daca obiect *clasa*, X se cauta prin mostenire
    - daca obiect *modul*, X este atribut direct al modulului

Spatii...

- Asignarea determina pozitia numelui/variabilei:

```
# Fisierul manynames.py:
X = 11 # Nume global/modul, nume/atribut: X,
      sau manynames.X
def f():
    print(X) # X = 11, e referinta globala
def g():
    X = 22 # X e local functiei (mascheaza X-ul
    din modul, global)
    print(X)
class C:
    X = 33 # Atribut de clasa (C.X)
    def m(self):
        X = 44 # Variabila locala metodei
        self.X = 55 # Atribut de instanta
    (C().X)
if __name__ == '__main__': # Autotestare
    print(X) # 11 din modul
    f() # 11: global
    g() # 22: local
    print(X) # 11: neschimbat in modul
    obj = C() # obj instanta a lui C
    print(obj.X) # 33: mostenit, din clasa
    obj.m() # X atasat instantei cu self.X = 55
    print(obj.X) # 55: gasit in instanta acum
    print(C.X) # 33: atributul clasei e neschimbat
    #print(C.m.X) # EROARE: nu e vizibil ca atribut de
    metoda
    #print(g.X) # EROARE: vizibil doar la apelul functiei
```

Spatii...

- Importarea face nume din module vizibile importatorului:

```
# Fisierul otherfile.py:
import manynames
X = 66
print(X) # 66: global aici
print(manynames.X) # 11: dupa importare
    globalele devin atribute de modul
manynames.f() # 11: f vede X-ul din
    manynames!
manynames.g() # 22: local functiei g
print(manynames.C.X) # 33: atribut de clasa din
    modulul celalalt
I = manynames.C() # instanta noua
print(I.X) # 33: mostenit din clasa
I.m()
print(I.X) # 55: atribut al instantei acum
```

Spatii...

- Efectul lui *global* si *nonlocal* in functii:

```
X = 11 # Global in modul

def g1():
    print(X) # Referinta globala, 11

def g2():
    global X
    X = 22 # Modifica X-ul global

def h1():
    X = 33 # Local in functie
    def nested():
        print(X) # Refera X-ul din functia h1, 33, regula conturului, LEGB

def h2():
    X = 3 # Local in functie
    def nested():
        nonlocal X # Python 3.X
        X = 44 # Modifica X-ul din functia h2
```

Spatii...

- <u>Regula conturului, LEGB, cu clase imbricate</u>
  - Clasele pot fi incluse in functii cu rol de fabrici de clase
    - clasa – si metodele sale au acces la nume cu LEGB – local, din functia unde sunt definite, din modul si *builtins*
    - <u>*clasa **nu** serveste ca spatiu de nume (de tip E) pentru metodele sale, asa incat **atributele clasei sunt accesate doar prin calificare si mostenire***</u>

```
X = 1 # Fisierul classscope.py
def nester():
    print(X) # Global: 1
    class C:  # Clasa inclusa in functia nester
        print(X) # Global: 1
        def method1(self):
            print(X) # Global: 1
        def method2(self):
            X = 3 # Globalul este mascat
            print(X) # Local: 3
    I = C()
    I.method1()
    I.method2()
print(X) # Global: 1
nester() # Afiseaza: 1, 1, 1, 3
print('-'*40)
```

Spatii...

- <u>Cazul lui X redefinit in functia *nester – mascheaza variabila globala*:</u>

```
X = 1
def nester():
    X = 2 # Mascheaza variabila globala
    print(X) # Local: 2
    class C:
        print(X) # Din nester: 2
        def method1(self):
            print(X) # Din nester: 2
        def method2(self):
            X = 3 # Mascheaza orice!
            print(X) # Local: 3
    I = C()
    I.method1()
    I.method2()
print(X) # Global: 1
nester() # Afiseaza: 2, 2, 2, 3
print('-'*40)
```

Spatii...

- Cazul lui X redefinit local in clase si functii – mascheaza X global si din functie:

```
X = 1
def nester():
    X = 2 # Mascheaza X-ul global
    print(X) # Local: 2
    class C:
        X = 3 # Mascheaza X-ul lui nester
        print(X) # Local: 3
        def method1(self):
            print(X) # X din nester (nu 3 din clasa!): 2
            print(self.X) # Mostenit de la clasa: 3
        def method2(self):
            X = 4 # Mascheaza pe X din nester (nu din clasa)
            print(X) # Local: 4
            self.X = 5 # Mascheaza X din clasa, acum este al instantei!
            print(self.X) # Din instanta: 5
    I = C()
    I.method1()
    I.method2()
print(X) # Global: 1
nester() # Afiseaza: 2, 3, 2, 3, 4, 5
print('-'*40)
```

Dictionarele spatiilor de nume

- Spatiile de nume ale modulelor, claselor si instantelor de clase sunt implementate ca dictionare referite de atributul predefinit ***__dict__***:

```
>>> class Super:
        def hello(self):
            self.data1 = 'spam'

>>> class Sub(Super):
        def hola(self):
            self.data2 = 'eggs'
```

- Atributele: **__dict__**, **__class__**, **__bases__**:

```
>>> X = Sub()
>>> X.__dict__ # __dict__ al instantei, vid initial
{}
>>> X.__class__ # Clasa instantei, atributul __class__
<class '__main__.Sub'>

>>> Sub.__bases__ # Tuplu cu superclase
(<class '__main__.Super'>,)
>>> Super.__bases__ # Tuplu vid in v2.X
(<class 'object'>,)
```

Dictionarele...

- Instantele pot fi diferite; exista si alte chei predefinite in dictionarele de clasa, e.g. **__doc__** (docstringuri):

```
>>> Y = Sub()
>>> X.hello() # Metoda mostenita de la Super
>>> X.__dict__
{'data1': 'spam'}
>>> X.hola() # Metoda de la clasa Sub
>>> X.__dict__
{'data2': 'eggs', 'data1': 'spam'}
```

```
>>> list(Sub.__dict__.keys())
['__module__', 'hola', '__doc__']
>>> list(Super.__dict__.keys())
['__module__', 'hello', '__dict__', '__weakref__',
    '__doc__']
>>> Y.__dict__
{}
```

Dictionarele...

- Atributele de instanta pot fi accesate si prin calificare si prin indexare in *__dict__*:

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')
>>> X.data3 = 'toast'
>>> X.__dict__
{'data2': 'eggs', 'data3': 'toast', 'data1': 'spam'}
>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

- *__dict__* -ul de instanta nu vede atributele din clasa!

```
>>> X.__dict__['hello']
KeyError: 'hello'
```

- ***dir(X)*** listeaza numele/cheile din instanta dar si multe din cele mostenite, sortate

Dictionarele...

- Vizualizarea arborelui cu spatii de nume al instantei:

```
#!python
"""
classtree.py: Afiseaza superclasele din ce in ce
    mai indentat
"""
def classtree(cls, indent):
    print('.' * indent + cls.__name__) # Afisarea
    numelui de clasa, indentat
    for supercls in cls.__bases__: # Iterare a
    superclaselor
            classtree(supercls, indent+3) # Apel
    recursiv
def instancetree(inst):
    print('Tree of %s' % inst) # Afisarea
    obiectului de tip instanta
    classtree(inst.__class__, 3) # Afisarea clasei
    instantei
def selftest():
    class A: pass
    class B(A): pass
    class C(A): pass
    class D(B,C): pass
    class E: pass
    class F(D,E): pass
    instancetree(B())
    instancetree(F())
if __name__ == '__main__':
    selftest()
```

Dictionarele...

- Output:

```
Tree of <__main__.selftest.<locals>.B object at 0x00000286984DAAC8>
...B
......A
.........object
Tree of <__main__.selftest.<locals>.F object at 0x00000286984DAAC8>
...F
......D
.........B
............A
...............object
.........C
............A
...............object
......E
.........object
```

Documentarea cu docstringuri

- Docstringurile se memoreaza in atributul __***doc***__ al modulelor, functiilor, claselor, metodelor:

```
# Fisierul docstr.py:
"I am: docstr.__doc__"
def func(args):
    "I am: docstr.func.__doc__"
    pass
class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__ or self.__doc__"
    def method(self):
        "I am: spam.method.__doc__ or self.method.__doc__"
        print(self.__doc__)
        print(self.method.__doc__)
```

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'
>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.spam.__doc__ or
    self.__doc__'
>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'
>>> x = docstr.spam()
>>> x.method()
I am: spam.__doc__ or docstr.spam.__doc__ or
    self.__doc__
I am: spam.method.__doc__ or self.method.__doc__
```

Documentarea...

- Formatarea docstringurilor se poate face cu **PyDoc**, iar afisarea in mod text, cu ***help()***:

```
>>> import docstr
>>> help(docstr)
Help on module __main__:

NAME
    __main__ - I am: docstr.__doc__

CLASSES
    builtins.object
        spam

    class spam(builtins.object)
 |  I am: spam.__doc__ or docstr.spam.__doc__ or self.__doc__
 |
 |  Methods defined here:
 |
 |  method(self)
 |      I am: spam.method.__doc__ or self.method.__doc__
 |
 |  ------------------------------------------------------------------
 |  Data descriptors defined here:
...etc...
```

Clase vs. module

- **Modulele**:
  - Implementeaza pachete de date, functii
  - Sunt create cu fisiere Python si extensii in alte limbaje
  - Sunt importate spre a fi folosite
  - Reprezinta nivelul cel mai inalt din structura codului Python
- **Clasele**:
  - Implementeaza obiectele in Python
  - Sunt create cu instructiunea ***class***
  - Sunt apelate spre a fi folosite
  - Se afla intr-un modul

Sumar

- Sintaxa claselor
- ***<u>Inlocuirea operatorilor</u>***

Notiuni de baza

- **Inlocuirea operatorilor – *operator overloading***, se refera la interceptarea operatiilor predefinite, cu metode, atunci cand instante ale claselor sunt prezente in expresii cu operatorii predefiniti inlocuiti
  - Rezultatul operatiei este returnat de metoda respectiva
  - Operatii normale in Python pot fi interceptate de clase
  - Clasele pot inlocui toti operatorii din expresii
  - Clasele pot inlocui operatii ca: afisarea, apelul de functii, selectia atributelor, etc
  - Inlocuirea produce clase cu comportament asemanator cu obiectele predefinite din Python
  - Inlocuirea se face in metode special numite

Notiuni...

- **Constructori si expresii, __*init*__ si __*sub*__:**
  - __*init*__ este costructorul clasei
  - __*sub*__ pentru scadere (*subtraction*)

```
# Fisier number.py:
class Number:
    def __init__(self, start): # Constructor
        self.data = start
    def __sub__(self, other): # Instanta - obiect
        return Number(self.data - other) # Returneaza o instanta noua
```

```
>>> from number import Number # Importarea clasei din modul
>>> X = Number(5) # Number.__init__(X, 5)
>>> Y = X - 2 # Number.__sub__(X, 2)
>>> Y.data # Y este noua instanta rezultat
3
```

Notiuni...

- Lista metodelor ce inlocuiesc operatori:

Metoda:	Implementeaza:	Apelata pentru:
__init__	Constructor	Creare obiect: X = Class(args)
__del__	Destructor	Garbage collector pentru X
__add__	Operatorul +	X + Y, X += Y daca nu exista __iadd__
__or__	Operatorul |	X | Y, X |= Y daca nu exista __ior__
__repr__, __str__	Print, conversii	print(X), repr(X), str(X)
__call__	Apel functie	X(*args, **kargs)
__getattr__	Citire atribut	X.undefined
__setattr__	Setare atribut	X.any = value
__delattr__	Stergere atribut	del X.any
__getattribute__	Citire atribut	X.any

Notiuni...

Metoda:	Implementeaza:	Apelata pentru:
__getitem__	Indexare, decupare, iteratie	X[key], X[i:j], cicluri (for) si alte iteratii daca nu exista __iter__
__setitem__	indexare/slice =	X[key] = value, X[i:j] = iterabil
__delitem__	Sterge index/slice	del X[key], del X[i:j]
__len__	Lungime	len(X), testare daca nu exista __bool__
__bool__	Test logic	bool(X), test adevar(__*nonzero*__ in v2.x)
__lt__, __gt__, __le__, __ge__, __eq__, __ne__	Comparatii	X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y (in lipsa __cmp__ numai in v2.X)
__iter__, __next__	Iteratii	I=iter(X), next(I); ciclu for, *in* daca nu exista __contains__, toate colectiile iterative, map(F,X), si altele (__next__ se numeste next in v2.X)

Notiuni...

Metoda:	Implementeaza:	Apelata pentru:
__index__	Valoare intreaga	hex(X), bin(X), oct(X), O[X], O[X:] (in loc de __oct__, __hex__ din v2.x)
__contains__	Apartenenta	item *in* X
__enter__,__exit__	Manager context	with obj as var:
__get__, __set__, __delete__	Atribute cu descriptori	X.attr, X.attr = value, del X.attr
__new__	Creare obiect	Apelat inainte de __init__

- Metodele speciale pot fi mostenite
- Sunt optionale in programare
- Nu sunt mai rapide decat expresiile normale, pot fi necesare sau elegante...

Indexare si decupare: __getitem__ si __setitem__

- Metodele sunt folosite pentru secvente si mapari
  - *__getitem__* este apelata pentru indexari ale instantei, e.g. **X[i]**:

```
>>> class Indexer:
        def __getitem__(self, index):
            return index ** 2
>>> X = Indexer()
>>> X[2] # X[i] apeleaza X.__getitem__(i)
4
```

```
>>> for i in range(5):
        print(X[i], end=' ') # Se executa Indexer.__getitem__(X, i) si afisare
0 1 4 9 16
```

- Decuparile se fac cu un obiect de tip:
  - **slice(start, stop[, step])**

```
>>> L = [5, 6, 7, 8]
>>> L[2:4], L[slice(2, 4)]
([7, 8], [7, 8])
>>> L[1:], L[slice(1, None)]
```

```
([6, 7, 8], [6, 7, 8])
>>> L[::2], L[slice(None, None, 2)]
([5, 7], [5, 7])
```

Indexare...

- ***__getitem__*** este apelata si pentru decupari/slicing:

```
>>> class Indexer:
        data = [5, 6, 7, 8, 9]
        def __getitem__(self, index): # Apelata pentru indexare si decupare
            print('getitem:', index)
            return self.data[index]
>>> X = Indexer()
>>> X[0] # Argument int
getitem: 0
5
>>> X[-1]
getitem: −1
9
```

```
>>> X[2:4] # Argument slice
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, −1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

Indexare...

- Varianta cu testarea tipului de argument (*int* sau *slice*):

```
>>> class Indexer:
        def __getitem__(self, index):
            if isinstance(index, int): # Testare tip argument, cu isinstance()
                print('indexing', index)
            else: # E slice daca nu e int
                print('slicing', index.start, index.stop, index.step)
```

```
>>> X = Indexer()
>>> X[99]
indexing 99
>>> X[1:99:2]
slicing 1 99 2
>>> X[1:]
slicing 1 None None
```

- ***__setitem__*** intercepteaza asignarile cu *int* sau *slice*:

```
class IndexSetter:
    def __setitem__(self, index, value): # Apelata pentru asignare de index sau decupaj/slice
        self.data[index] = value
```

Indexare...

- Metoda ___***index***___ nu indexseaza!
  - Returneaza o valoare intreaga in cazul folosirii de operatori predefiniti care convertesc un *int* la *str*-ul cifrelor intr-o baza de numeratie:

```
>>> class C:
        def __index__(self):
            return 255
>>> X = C()
>>> hex(X) # hex(int) => str hexa
'0xff'
>>> bin(X) # bin(int) => str binar
'0b11111111'
>>> oct(X) # oct(int) => str octal
'0o377'
```

  - Se foloseste si cand un intreg este necesar e.g. pentru expresia indiciala:

```
>>> ('C' * 256)[X] #
'C'
>>> ('C' * 256)[X:]
'C'
```

Iteratie cu __*getitem*__

- Este posibila, intr-un ciclu ***for*** si in contexte iterative
  - Se indexeaza repetat de la 0, 1, ... pana se produce exceptia *IndexError*.

```
>>> class StepperIndex:
        def __getitem__(self, i):
            return self.data[i]
>>> X = StepperIndex() # X instanta a StepperIndex
>>> X.data = "Spam"
```

```
>>> X[1] # Indexare cu __getitem__
'p'
>>> for item in X: # Instructiunea for apeleaza __getitem__
        print(item, end=' ')
S p a m
```

```
>>> 'p' in X # in, iteratii, toate cu __getitem__
True
>>> [c for c in X] # Lista iterativa
['S', 'p', 'a', 'm']
>>> list(map(str.upper, X)) # map()
['S', 'P', 'A', 'M']
```

```
>>> (a, b, c, d) = X # Asignare de secventa
>>> a, c, d
('S', 'a', 'm')
>>> list(X), tuple(X), ''.join(X) # Etc...
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')
```

Obiecte iterabile cu __*iter*__ si __*next*__

- **Protocolul iterativ** are precedenta fata de indexarea repetata cu __*getitem*__
  - Metoda __***iter***__ returneaza un obiect iterator **I**, apelata fiind cu functia predefinita *iter(instanta)*
  - Metoda __***next***__ a iteratorului **I** produce obiecte pana cand se produce exceptia *StopIteration*
    - Apelata cu functia predefinita ***next(I)***, este echivalenta cu ***I.__next__()***

```
# Fisier squares.py:
class Squares:
    def __init__(self, start, stop): # Constructor
        self.value = start - 1
        self.stop = stop
    def __iter__(self): # Returneaza obiectul iterator, self!
        return self
    def __next__(self): # Are si metoda __next__, deci clasa este si iterator
        if self.value == self.stop:
            raise StopIteration # Gata!
        self.value += 1 # Incrementare
        return self.value ** 2 # Ridicare la patrat
```

Obiecte...

- Context iterativ, automat:

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

>>> **from squares import Squares**

>>> **for i in Squares(1, 5):** *# Instanta, for apeleaza pe __iter__*

print(i, end=' ') *# Fiecare iteratie apeleaza pe __next__*

1 4 9 16 25

- Iteratie manuala:

>>> **X = Squares(1, 5)**

>>> **I = iter(X)** *# iter(X) este X.__iter__() sau Squares.__iter__(X); I este instanta a lui Squares!*

>>> **next(I)** *# next(I) este I.__next__() sau Squares.__next__(I)*

1

>>> **next(I)**

4

...etc...

>>> **next(I)**

25

>>> **next(I)** *# Exceptia se poate intercepta cu* ***try:***

StopIteration

Obiecte...

- Indexarea nu este suportata, doar iteratia:

```
>>> X = Squares(1, 5)
>>> X[2] # Indexare, eroare!
TypeError: 'Squares' object is not subscriptable
>>> list(X)[2] # Iteratie completa + indexare list
9
>>> list(X)[2] # Nu suporta iteratori multipli!
IndexError: list index out of range
>>> X = Squares(1, 5) # Iterator nou
>>> list(X)[2] # OK!
9
```

- <u>Scanari multiple necesita iteratori noi</u>:

```
>>> X = Squares(1, 5) # Iterator nou
>>> [n for n in X] # Iteratie completa
[1, 4, 9, 16, 25]
>>> [n for n in X] # Nimic, iteratorul a fost
    epuizat deja
[]
>>> 36 in Squares(1, 10) # Iteratie cu in
True
>>> a, b, c = Squares(1, 3) # Iteratie cu asignare
    de tuple
>>> a, b, c
(1, 4, 9)
>>> ':'.join(map(str, Squares(1, 5))) # Iteratie cu
    join si map
'1:4:9:16:25'
```

Obiecte...

- Dupa conversia la *list*, se pot face oricate iteratii:

```
>>> X = Squares(1, 5)
>>> tuple(X), tuple(X) # Iterator epuizat la al doilea tuple(X)
((1, 4, 9, 16, 25), ())
```

```
>>> X = list(Squares(1, 5)) # O iteratie=>list
>>> tuple(X), tuple(X) # Oricate iteratii pe list!
((1, 4, 9, 16, 25), (1, 4, 9, 16, 25))
```

- Functiile sau expresiile generator itereaza mai simplu decat clasele:

```
>>> def gsquares(start, stop): # Functie generator
        for i in range(start, stop + 1):
            yield i ** 2
>>> for i in gsquares(1, 5): # Iterare cu for a functiei
        print(i, end=' ')
1 4 9 16 25
```

```
>>> for i in (x ** 2 for x in range(1, 6)): # Expresie generator iterata cu for
        print(i, end=' ')
1 4 9 16 25
>>> [x ** 2 for x in range(1, 6)] # Cel mai simplu, colectia iterativa!
[1, 4, 9, 16, 25]
```

Obiecte...

- Obiecte cu iteratori multipli (independenti):
  - Iteratia multipla este suportata de obiecte predefinite e.g. *list*, *str* si de functia predefinita *range()*
  - *map* si *zip* sunt iteratori unici

```
>>> S = 'ace'
>>> for x in S: # Iteratie multipla cu str
        for y in S:
            print(x + y, end=' ')
aa ac ae ca cc ce ea ec ee
```

  - Clase cu iteratori multipli:

```
#!python
# Fisier skipper.py
class SkipObject:
    def __init__(self, wrapped): # Constructor
        self.wrapped = wrapped
    def __iter__(self):
        return SkipIterator(self.wrapped) # Returneaza o instanta noua de SkipIterator care este un iterator – are metoda __next__
```

Obiecte...

```
class SkipIterator:
    def __init__(self, wrapped): # Constructor
        self.wrapped = wrapped # Obiectul de parcurs, diferit per instanta
        self.offset = 0 Pozitia curenta, diferita per instanta
    def __next__(self): # Are __next__, deci este iterator
        if self.offset >= len(self.wrapped):
            raise StopIteration # Gata!!
        # Returneaza item, avanseaza
        item = self.wrapped[self.offset]
        self.offset += 2
        return item
```

```
if __name__ == '__main__': # Autotestare
    alpha = 'abcdef'
    skipper = SkipObject(alpha) # Instanta
    I = iter(skipper) # skipper returneaza un obiect iterator
    print(next(I), next(I), next(I)) # Pozitiile 0, 2, 4
    for x in skipper: # for apeleaza __iter__ automat
        for y in skipper: # for imbricat care apeleaza __iter__ din nou
            print(x + y, end=' ') # Fiecare iterator are pozitie independenta
```

```
C:\code> py -3 skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

Obiecte...

- Clase vs. decupaj/slice:

```
>>> S = 'abcdef'
>>> for x in S[::2]:
        for y in S[::2]: # Obiecte noi,decupate
            print(x + y, end=' ')
aa ac ae ca cc ce ea ec ee
```

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
        for y in S: # Acelas ob., iteratori noi
            print(x + y, end=' ')
aa ac ae ca cc ce ea ec ee
```

- Clasele pot defini iteratori utili in diverse aplicatii:
  - e.g. baze de date, cu scanari independente ale rezultatului unei cereri SQL

Obiecte...

- Alternativa pentru iteratia multipla, cu **__iter__** si **yield**:
  - Functiile generator (cu *yield*) produc obiecte iterabile:

```
>>> def gen(x):
        for i in range(x): yield i ** 2
>>> G = gen(5) # Generator, are __iter__ si __next__
>>> G.__iter__() == G # G este si iterator
True
>>> I = iter(G) # Apel de __iter__
>>> next(I), next(I) # Apel de __next__ (next in v2.X)
(0, 1)
>>> list(gen(5)) # Context iterativ automat
[0, 1, 4, 9, 16]
```

  - Clasa cu __***iter__/yield***:
    - Metoda __*next*__ este implicita (lipseste, corect):

```
# Fisier squares_yield.py:
class Squares: # __iter__/yield
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
    def __iter__(self): # Returneaza obiect generator nou, care are __iter__ si __next__
        for value in range(self.start, self.stop + 1):
            yield value ** 2
```

Obiecte...

- Output:

C:\code> py **-3** >>> **from squares_yield import Squares**	>>> **for i in Squares(1, 5): print(i, end=' ')** 1 4 9 16 25

>>> *# Iteratie manuala*
>>> **S = Squares(1, 5)**
>>> **S**
<__main__.Squares object at 0x0000023F1DD5B1C8>
>>> **I = iter(S)** *# Apeleaza pe S.__iter__() automat, rezultat obiect generator I*
>>> **I == I.__iter__()** *# Obiect iterabil, este si propriul sau iterator*
True
	>>> I
	<generator object Squares.__iter__ at 0x0000023F1DD40D48>
	>>> **next(I)** *# Apeleaza pe I.__next__()*
	1
	...etc...
	>>> **next(I)**
	25
	>>> **next(I)**
	StopIteration

Obiecte...

- Metoda cu *yield* (__iter__) returneaza un obiect generator nou, independent, la fiecare apel:

```
C:\code> py -3
>>> from squares_yield import Squares
>>> S = Squares(1, 5)
>>> I = iter(S)
>>> next(I), next(I)
(1, 4)
>>> J = iter(S) # Alt generator
>>> next(J)
1
>>> next(I) # I este independent de J
9
>>> # Iteratii independente:
>>> S = Squares(1, 3)
>>> for i in S: # Fiecare for apeleaza __iter__
        for j in S:
            print('%s:%s' % (i, j), end=' ')
1:1 1:4 1:9 4:1 4:4 4:9 9:1 9:4 9:9
```

Obiecte...

- ***Squares*** fara *yield*, dar cu doua clase, iterator multiplu:

```
class Squares: # Generator fara yield, 2 clase
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
    def __iter__(self):
        return SquaresIter(self.start, self.stop)
class SquaresIter:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop = stop
    def __next__(self):
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

- ***SkipObject*** cu *yield*, iterator multiplu:

```
class SkipObject:
    def __init__(self, wrapped): # Constructor
        self.wrapped = wrapped
    def __iter__(self):
        offset = 0
        while offset < len(self.wrapped):
            item = self.wrapped[offset]
            offset += 2
            yield item
```

Apartenenta cu __*contains*__

- Metoda __**contains**__ este preferata fata de __*iter*__ care este preferat fata de __*getitem*__

```
# Fisier contains.py:
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i): # Rezerva, pt. iteratie, indexare, slice
        print('get[%s]:' % i, end='')
        return self.data[i]
    def __iter__(self): # Preferata pt. iteratie
        print('iter=> ', end='') # Doar un singur iterator
        self.ix = 0
        return self
    def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data): raise StopIteration # Gata!!
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x): # Preferata pt. 'in'
        print('contains: ', end='')
        return x in self.data
    next = __next__ # Asignare in clasa, compatibilitate cu v2.x
```

Apartenenta...

- Clasa ***Iters*** cu *yield*, mai simplu (fara __*next*__):

```
# Fisier contains_yield.py:
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i): # Rezerva, pt. iteratie, indexare, slice
        print('get[%s]:' % i, end='')
        return self.data[i]
    def __iter__(self): # Preferata pt. iteratie
        print('iter=> next:', end='') # Iteratori multipli, simultan
        for x in self.data:
            yield x
            print('next:', end='')
    def __contains__(self, x): # Preferata pt. 'in'
        print('contains: ', end='')
        return x in self.data
# Observatie: print e doar pentru trasare...
```

Selectia atributelor cu __***getattr***__, __***setattr***__ si __**delattr**__

- Selectia – referirea, asignarea, stergerea (cu *del*), avand sintaxa ***obiect.atribut*** se poate implementa cu clase:
  - __***getattr***__ intercepteaza referirea la atribute, le calculeaza o valoare in mod dinamic, produce exceptii pentru cele nesuportate, cu *raise*

```
>>> class Empty:
        def __getattr__(self, attrname):
            if attrname == 'age': return 40
            raise AttributeError(attrname)
```

```
>>> X = Empty() # Instanta nu are atribute dar...
>>> X.age # instanta.atribut => se executa
      __getattr__=> atribut dinamic, la executie
40
>>> X.name # Se produce exceptie pentru
      atributele nesuportate
AttributeError: name
```

Selectia...

- ***__setattr__*** intercepteaza toate asignarile de atribute:
  - Expresia ***self.atr = val*** devine ***self.__setattr__('atr', val)***
  - Pericolul recursivitatii infinite: in metoda *__setattr__* orice expresie *self.atribut* = *valoare* apeleaza din nou *__setattr__*
    - ***self.__dict__['name'] = valoare*** evita recursivitatea – indexare de *dict*, nu simpla setare de atribut!

```
>>> class Accesscontrol:
        def __setattr__(self, attr, value):
            if attr == 'age':
                self.__dict__[attr] = value + 10 # NU self.name=val sau setattr(self, attr, val)
            else:
                raise AttributeError(attr + ' not allowed')
```

```
>>> X = Accesscontrol()
>>> X.age = 40 # Apel de __setattr__
>>> X.age
50
>>> X.name = 'Bob'
AttributeError: name not allowed
```

Selectia...

>>> ***# Gresit:***	>>> ***# Corect:***
self.age = value + 10 *# Recursivitate infinita*	**self.__dict__[attr] = value + 10**
setattr(self, attr, value + 10) *# Recursivitate infinita cand attr este 'age'*	**object.__setattr__(self, attr, value + 10)** *# Cu superclasa object!*
self.other = 99 *# Rateaza cu AttributeError: other not allowed*	

- __***delattr***__ intercepteaza stergerea de atribute, e.g. ***del obiect.atribut***
  - are ca argument stringul nume de atribut
  - recursivitatea se evita cu __dict__ sau superclasa object

- Alte metode pentru atribute:
  - __***getattribute***__ intercepteaza toate referirile, nu numai cele nedefinite
  - Cu functia predefinita ***property(),*** cu ***descriptori***, cu ***slots***

Selectia...

- Emularea atributelor private:

```
class PrivateExc(Exception): pass # Exceptii, urmeaza
class Privacy:
    def __setattr__(self, attrname, value): # Intercepteaza self.attrname = value
        if attrname in self.privates: # Delegatie!
            raise PrivateExc(attrname, self) # Produce o exceptie definita de utilizator
        self.__dict__[attrname] = value
class Test1(Privacy):
    privates = ['age']
class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'
if __name__ == '__main__': # Autotestare
    x = Test1()
    y = Test2()
    x.name = 'Bob' # OK
    #y.name = 'Sue' # Exceptie
    print(x.name)
    y.age = 30 # OK
    #x.age = 40 # Exceptie
    print(y.age)
# Cu decoratori, urmeaza...
```

Reprezentarea stringurilor cu __*repr*__ si __*str*__

- Codificare cu metoda __*repr*__:

```
>>> class adder: # Aduna
        def __init__(self, value=0):
            self.data = value
        def __add__(self, other):
            self.data += other

>>> class addrepr(adder): # Mosteneste __init__, __add__
    def __repr__(self): # Reprezentare a str
        return 'addrepr(%s)' % self.data # Conversie la un str
```

```
>>> x = addrepr(2) # Apel de __init__ din adder
>>> x + 1 # Apel de __add__
>>> x # Apel de __repr__
addrepr(3)
>>> print(x) # Apel de __repr__
addrepr(3)
>>> str(x), repr(x) # Apel de __repr__ pentru ambele
('addrepr(3)', 'addrepr(3)')
```

- Atat __*repr*__ cat si __*str*__ se folosesc la conversia instantelor la **string**

Reprezentarea...

- Rolul celor doua metode de afisare:
  - **__str__** este incercata prima de *print()* si *str()*
    - reprezentare *user-friendly*
  - **__repr__** se foloseste in sesiuni interactive sau daca *__str__* lipseste – si de *print()*
    - reprezentare – cod, pentru recrearea obiectelor sau cu detalii pentru programatori

```
>>> class addstr(addrepr): # Adaugare __str__
        def __str__(self):
            return '[Value: %s]' % self.data
    # Conversie la un str user-friendly
>>> x = addstr(3)
>>> x + 1
>>> x
addrepr(4)
```

```
>>> print(x)
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', 'addrepr(4)')
```

Reprezentarea...

- Se recomanda inlocuirea numai a lui __*repr*__, nu si a lui __*str*__ cand instantele de afisat sunt incluse in alt obiect, e.g. o lista:

```
>>> class Printer:
        def __init__(self, val):
            self.val = val
        def __repr__(self): # __repr__ e folosit de print daca nu exista __str__
            return str(self.val) # __repr__ este folosita si pentru obiecte imbricate
```

```
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x) # Apel de __repr__
2
3
>>> print(objs) # Apel de __repr__, nu de __str__
[2, 3]
>>> objs # Apel de __repr__
[2, 3]
```

In dreapta si in-place cu __*radd*__ si __*iadd*__

- Adunare cu instante aflate in dreapta simbolului **+**:
  - Problema cu __*add*__*:*

```
>>> class Adder:
    def __init__(self, value=0):
        self.data = value
    def __add__(self, other):
        return self.data + other
```

```
>>> x = Adder(5)
>>> x + 2 # OK cu x in stanga lui +
7
>>> 2 + x # x este in dreapta...
TypeError: unsupported operand type(s) for +:
    'int' and 'Adder'
```

  - Comutativitate suportata cu __***radd***__ si __*add*__:

```
>>> class Commuter1:
        def __init__(self, val):
            self.val = val
        def __add__(self, other):
            print('add', self.val, other)
            return self.val + other
# noninstanta + instanta? => apel de __radd__
        def __radd__(self, other):
            print('radd', self.val, other)
            return other + self.val
```

In..

```
>>> x = Commuter1(88)
>>> y = Commuter1(99)
>>> x + 1 # instanta + noninstanta, __add__
add 88 1
89
>>> 1 + y # noninstanta + instanta, __radd__
radd 99 1
100
>>> x + y # instanta + instanta, __add__ => noninstanta + instanta, __radd__ !!
add 88 <__main__.Commuter1 object at 0x0000022CA0733188>
radd 99 88
187
```

- **Refolosirea lui __*add*__ in __*radd*__:**
  - ➢ Cu apel direct, adunare cu schimbarea ordinii pentru a declansa __add__, facand __radd__ un alias pentru __add__

```
class Commuter2: # Cu apel explicit de add
    def __init__(self, val):
        self.val = val
    def __radd__(self, other):
        return self.__add__(other)
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
```

In..

```
class Commuter3: # Adunare cu schimbarea ordinii
    def __init__(self, val):
        self.val = val
    def __radd__(self, other):
        return self + other
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
```

```
class Commuter4: # Cu alias, simplu!
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    __radd__ = __add__
```

In..

- Propagarea tipului clasei in rezultate:

```
class Commuter5:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        if isinstance(other, Commuter5): # Se evita imbricarea obiectelor!!
            other = other.val # other devine valoare, nu mai este clasa
        return Commuter5(self.val + other)
    def __radd__(self, other):
        return Commuter5(other + self.val)
    def __str__(self):
        return '<Commuter5: %s>' % self.val
```

```
>>> x = Commuter5(88)
>>> y = Commuter5(99)
>>> print(x + 10) # Instanta a clasei Commuter5
<Commuter5: 98>
>>> print(10 + y)
<Commuter5: 109>
>>> z = x + y # Se evita __radd__
>>> print(z)
<Commuter5: 187>
>>> print(z + 10)
<Commuter5: 197>
>>> print(z + z)
<Commuter5: 374>
>>> print(z + z + 1)
<Commuter5: 375>
```

In..

- Adunare in-place, cu **__iadd__**, pentru operatorul **+=**:

```
>>> class Number:
        def __init__(self, val):
            self.val = val
        def __iadd__(self, other): # __iadd__ explicit: x += y
            self.val += other
            return self

>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
```

  - Cazul obiectelor modificabile:

```
>>> y = Number( [1] ) # Mai rapid decat adunarea
>>> y += [2]
>>> y += [3]
>>> y.val
[1, 2, 3]
```

In..

- *__add__* este de rezerva pentru += (cand __iadd__ lipseste):

```
>>> class Number:
        def __init__(self, val):
            self.val = val
        def __add__(self, other): # __add__ calculeaza: x = (x + y)
            return Number(self.val + other)  # Tipul clasei este propagat
```

```
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
# Atentie x+y inseamna concatenare!
```

- De notat ca toti operatorii binari au versiuni de dreapta si in-place, e.g. ***__mul__***, ***__rmul__***, ***__imul__*** pentru inmultire

Apelul cu __*call*__

- Metoda __***call***__ este apelata cand <u>instanta</u> apare intr-o expresie de tip apel de functie, cu oricate argumente pozitionale sau cu cuvinte cheie:

```
>>> class Callee:
        def __call__(self, *pargs, **kargs): # Intercepteaza apeluri
            print('Called:', pargs, kargs) # Accepta argumente arbitrare
>>> C = Callee()
>>> C(1, 2, 3) # Instanta C este un obiect apelabil
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

- Metoda preia toate tipurile de argumente de functii:

```
class C:
    def __call__(self, a, b, c=5, d=6): ... # Arg. normale si implicite
class C:
    def __call__(self, *pargs, **kargs): ... # Arg. arbitrare
class C:
    def __call__(self, *pargs, d=6, **kargs): ... # Arg. numai cu cuvinte cheie din v3.x
```

Apelul...

```
X = C()
X(1, 2) # Valori implicite omise
X(1, 2, 3, 4) # Pozitional
X(a=1, b=2, d=4) # Cu cuvinte cheie
X(*[1, 2], **dict(c=3, d=4)) # Despachetare cu * si **
X(1, *(2,), c=3, **dict(d=4)) # Mod mix
```

- Alt exemplu:

```
>>> class Prod:
        def __init__(self, value): # Un argument
            self.value = value
        def __call__(self, other):
            return self.value * other
>>> x = Prod(2) # 2 este retinut, stare
>>> x(3) # 3 (actual) * 2 (stare)
6
>>> x(4)
8
```

- Metoda *__call__* este utila la interfatarea cu un API dat
  - Dupa *__init__*, *__str__*/*__repr__*, ***__call__*** este, ca frecventa, a treia metoda ce se inlocuieste in Python

Apelul...

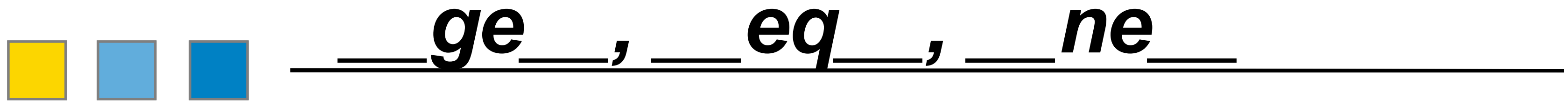
- Interfete de functii si cod tip *callback*, cu tkinter/GUI:

```
class Callback:
    def __init__(self, color): # color este informatie de stare
        self.color = color
    def __call__(self): # Apel fara argumente
        print('turn', self.color)
```

```
# Instante ale clasei sunt comenzi efectuate la apasarea butoanelor:
cb1 = Callback('blue') # blue este memorat
cb2 = Callback('green') # green este memorat
B1 = Button(command=cb1) # Inregistrare comenzi la crearea butoanelor
B2 = Button(command=cb2)

# Evenimente, apasare de butoane:
cb1() # Afiseaza 'turn blue'
cb2() # Afiseaza 'turn green'
```

Comparatii cu __lt__, __gt__, __le__, __ge__, __eq__, __ne__



- Se intercepteaza operatorii de comparatie: **<, >, <=, >=, ==** si **!=**
  - Nu exista operatori de dreapta
  - __**eq**__ si __**ne**__ trebuie definiti impreuna
  - Numai in Python v2.x exista metoda __**cmp**__

```
class C:
    data = 'spam'
    def __gt__(self, other): # v3.x si v2.x
        return self.data > other
    def __lt__(self, other):
        return self.data < other
```

```
>>> X = C()
>>> print(X > 'ham')
True
>>> print(X < 'ham')
False
```

Teste logice cu __*bool*__ si __*len*__

- Se determina valoarea de adevar a obiectelor, intai cu metoda __*bool*__, iar daca lipseste, cu __*len*__:

```
>>> class Truth:
        def __bool__(self): return True
>>> X = Truth()
>>> if X: print('yes!')
yes!
```

```
>>> class Truth:
        def __bool__(self): return False
>>> X = Truth()
>>> bool(X)
False
```

```
>>> class Truth: # Lungime zero inseamna False
        def __len__(self): return 0
>>> X = Truth()
>>> if not X: print('no!')
no!
```

```
>>> class Truth: pass # Daca ambii lipsesc True
>>> X =  Truth()
>>> bool( X )
True
```

```
>>> class Truth: # Daca ambii prezenti __bool__ are prioritate in v3.x, viceversa in v2.x
        def __bool__(self): return True
        def __len__(self): return 0
>>> bool( Truth() )
True
```

Stergerea obiectelor cu __***del***__

- __*new*__ se executa inainte de __init__ (se creeaza instanta), apoi __*init*__
- __***del***__ se executa cand spatiul de memorie al istantei este eliberat (de *garbage collector*)

```
>>> class Life:
        def __init__(self, name='unknown'):
            print('Hello ' + name)
            self.name = name
        def live(self):
            print(self.name)
        def __del__(self):
            print('Goodbye ' + self.name)
```

```
>>> brian = Life('Brian')
Hello Brian
>>> brian.live()
Brian
>>> brian = 'loretta'
Goodbye Brian
```

Stergerea...

- Utilitatea lui __*del*__:
  - Este redusa in Python, fiindca exista mecanismul de *garbage collection*
  - Nu se poate determina cu precizie cand un obiect este sters (mai poate fi referit din alte parti ale programului)
  - Exceptiile produse in codul lui __*del*__ doar scriu un mesaj la *sys.stderr*
  - Obiecte referite circular pot fi sterse numai daca **nu** au metoda __del___