Programarea calculatoarelor si limbaje de programare II

Exceptii

Universitatea Politehnica din Bucureşti

Sumar

- ***Rolul exceptiilor***
- Codificare cu exceptii
- Instructiunea *try/except/else*
- Instructiunea *try/finally*
- Instructiunea *try/except/finally*
- Instructiunea *raise*
- Instructiunea *assert*
- Manageri de context cu *with/as*

Notiuni de baza

- Exceptiile in Python:
  - Permit modificarea ordinii de executie dintr-un program
  - Sunt declansate automat la producerea de erori
  - Pot fi declansate si interceptate in codul utilizatorului
- Instructiuni de procesare a exceptiilor:
  - **try/except** – interceptare si tratare a exceptiilor declansate de Python sau in codul nostru
  - **try/finally** – operatii efectuate, daca se produc exceptii sau nu
  - **raise** – declanseaza o exceptie in codul Python
  - **assert** – declanseaza o exceptie daca o conditie este falsa
  - **with/as** – implementeaza manageri de context

Rolul exceptiilor

- Exceptia permite saltul la un cod de tratare a exceptiei, abandonandu-se toate apelurile de functii ulterioare clauzei *try.* Se revine la starea de dinainte de *try*, urmand tratarea erorii
- Rolul exceptiilor in Python:
  - **Tratarea erorilor** – netratate, incheie executia cu un mesaj
  - **Notificarea evenimentelor** – e.g. declansarea unei exceptii in loc de returnarea unui rezultat cu o valoare anume
  - **Codificarea cazurilor speciale** – pentru conditii rare care nu justifica verificari multiple, cu exceptie in loc
  - **Incheierea actiunilor** – cu un cod ce se executa chiar daca a avut loc o exceptie (cu *try/finally*)
  - **Emularea saltului** – cu *raise*, din interiorul mai multor instructiuni repetitive

Sumar

- Rolul exceptiilor
- ***Codificare cu exceptii***
- Instructiunea *try/except/else*
- Instructiunea *try/finally*
- Instructiunea *try/except/finally*
- Instructiunea *raise*
- Instructiunea *assert*
- Manageri de context cu *with/as*

Tratamentul implicit al erorilor

- Se afiseaza un mesaj de eroare, care indica eroarea produsa si include si informatii despre stiva de executie curenta, iar un script Python este terminat:

```
>>> def fetcher(obj, index):
...         return obj[index]
...
>>> x = 'spam'
>>> fetcher(x, 3) # Ca x[3]
'm'
```

```
>>> fetcher(x, 4) # Implicit – in terminal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

```
>>> def fetcher(obj, index):
        return obj[index]
>>> x = 'spam'
>>> fetcher(x, 4) # Implicit – cu IDLE
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    fetcher(x, 4) # Default handler - IDLE GUI
      interface
  File "<pyshell#2>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

Interceptarea exceptiilor

- Se face prin includerea portiunii de cod susceptibila de producerea unei erori in instructiunea compusa ***try***, a carei clauza ***except*** cuprinde codul de tratare a erorii:

```
try:
    fetcher(x, 4)
except IndexError: # Numele exceptiei de interceptat
    print('got exception')
got exception
```

```
def catcher():
    try:
        fetcher(x, 4)
    except IndexError:
        print('got exception')
    print('continuing')
>>> catcher()
got exception
continuing
```

- Programul continua dupa instructiunea *try*, de tratare a erorii

Declansarea exceptiilor

- Se face cu instructiunea ***raise***, din program, iar interceptia este la fel ca atunci cand Python declanseaza exceptia:

```
>>> try:
        raise IndexError # Exceptie declansata manual
except IndexError:
        print('got exception')
got exception
```

```
>>> raise IndexError # Exceptie tratata implicit
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    raise IndexError
IndexError
```

- Exceptie declansata cu instructiunea ***assert***:

```
>>> assert False, 'Nobody expected the coronavirus!'
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    assert False, 'Nobody expected the coronavirus!'
AssertionError: Nobody expected the coronavirus!
```

Exceptii definite de utilizator

- Sunt create cu clase, de obicei derivate din clasa predefinita ***Exception***:

```
>>> class AlreadyGotOne(Exception): pass # Exceptie definita de utilizator
>>> def grail():
        raise AlreadyGotOne() # Declanseaza o instanta a clasei
```

```
>>> try:
        grail()
except AlreadyGotOne: # Interceptie cu numele clasei
        print('got exception')
got exception
```

```
>>> class Career(Exception): # Cu mesaj de eroare ales de utilizator
        def __str__(self): return 'So I became a waiter...'
>>> raise Career()
```

```
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    raise Career()
Career: So I became a waiter...
```

Cod final

- Cu clauza ***finally*** se poate preciza cod care se va executa intotdeauna, cand fie se produce fie nu se produce exceptia:

```
>>> try: # Eventuala eroare netratata se propaga la tratamentul implicit
        fetcher(x, 7)
finally:
        print('after fetch')
after fetch
IndexError: string index out of range
>>> try:
        fetcher(x, 7)
except IndexError: # Eroare captata
        print('Index error!!')
finally:
        print('after fetch')
Index error!!
after fetch
>>> try: # Fara eroare
        fetcher(x, 3)
finally:
        print('after fetch')
'm'
after fetch
```

Cod...

- O eroare netratata se propaga la un *try* anterior sau la tratamentul implicit, dar blocul *finally* se executa inainte de propagare:

```
>>> def after():
        try:
            fetcher(x, 7)
        finally:
            print('after fetch')
        print('after try?')
>>> after()
after fetch
IndexError: string index out of range
```

```
>>> def after():
        try:
            fetcher(x, 3)
        finally:
            print('after fetch')
        print('after try?')
>>> after()
after fetch
after try?
```

Cod...

- Managementul de context se poate face cu instructiunea ***with/as***:

```
>>> # Fisierul va fi intotdeauna inchis
>>> with open('lumberjack.txt', 'w') as file:
        file.write('The larch!\n')
11
>>> import os
>>> for i in os.popen( 'dir lumberjack.txt' ):
        print(i, end='')


 Volume in drive C has no label.
 Volume Serial Number is 1C54-9759

 Directory of c:\Users\Dan\Desktop

03/20/2020  04:40 PM                12 lumberjack.txt
               1 File(s)             12 bytes
               0 Dir(s)  180,123,619,328 bytes free
```

Sumar

- Rolul exceptiilor
- Codificare cu exceptii
- ***Instructiunea try/except/else***
- Instructiunea *try/finally*
- Instructiunea *try/except/finally*
- Instructiunea *raise*
- Instructiunea *assert*
- Manageri de context cu *with/as*

Instructiunea *try/except/else*

- Incepand cu Python v2.5, clauza ***finally*** poate fi de asemenea prezenta. Sintaxa:

try:

statements # Cod de executat mai intai

except *name1*:

statements # Cod de executat daca exceptia *name1* se produce

except (*name2*, *name3*):

statements # Cod de executat daca exceptiile *name2* sau *name3* se produc

except *name4* **as** *var*:

statements # Cod de executat daca exceptia *name4* se produce, var = instanta

except:

statements # Cod de executat pentru toate celelalte exceptii

else:

statements # Cod de executat daca nici o exceptie **nu** se produce

Instructiunea...

- Executia:
  - Se memoreaza contextul programului la inceputul lui *try*
  - Se executa instructiunile din blocul *try*
  - Daca se produce o exceptie desemnata in *try,* atunci se reconstituie contextul lui *try* si se executa instructiunile corespunzatoare exceptiei dupa ce in prealabil se asigneaza obiectul exceptie variabilei din clauza *as* – daca este prezenta. Apoi executia continua dupa intreaga instructiune *try*.
  - Daca se produce o exceptie diferita, atunci exceptia este propagata catre o instructiune *try* anterioara care o poate trata. In lipsa, se ajunge la nivelul cel mai de sus al programului, unde Python afiseaza un mesaj de eroare si termina executia.
  - Daca nu se produce nici o exceptie, se executa instructiunile din blocul *else*, daca prezent, si apoi se continua dupa intreaga instructiune *try*.

Instructiunea...

- Clauzele din instructiunea ***try***:
  - Oricate clauze *except*, dar cel putin una asa incat clauza *else* sa fie permisa
  - O singura clauza *finally* si o singura clauza *else*

Clauza	Rol
except:	Intercepteaza toate (celelalte) exceptii
except *name*:	Intercepteaza o singura exceptie
except *name* as *value*:	Intercepteaza exceptia si o asigneaza
except (*name1*, *name2*):	Intercepteaza exceptiile listate
except (*name1*, *name2*) as *value*:	Intercepteaza exceptiile si o asigneaza
else:	Se executa daca nici o exceptie
finally:	Se executa intotdeauna

Instructiunea...

- Interceptarea oricarei exceptii listate si a tuturor exceptiilor:
  - ***except (e1, e2, e3):*** se intercepteaza oricare dintre exceptiile listate
  - daca exceptia nu este tratata atunci se propaga la un *try* inconjurator sau este tratata implicit (mesaj+terminare)

```
try:
    action()
except NameError:
    ...
except IndexError:
    ...
except KeyError:
    ...
except (AttributeError, TypeError, SyntaxError):
    ...
else: ... # Nici o exceptie!
```

Instructiunea...

- ***except:*** se intercepteaza toate exceptiile nelistate anterior in *try*
- Interceptarea oricarei exceptii:

```
try:
    action()
except NameError: # Trateaza NameError
    ...
except IndexError: # Trateaza IndexError
    ...
except: # Trateaza toate celelalte exceptii
    ...
else: # Cazul nici unei exceptii
    ...
```

```
try:
    action()
except: # Tratarea oricarei exceptii
    ...

try:
    action()
except Exception: # Se trateaza orice
exceptie in afara de iesirea din program
(e.g. Ctrl/C)
    ...
```

Instructiunea...

- Clauza *try* ***else***:
  - Rezolva ambiguitatile:

```
try:
    ...executa cod...
except IndexError:
    ...trateaza exceptia...
# Oare try a ratat sau nu ?
```

```
try:
    ...executa cod...
except IndexError:
    ...trateaza exceptia...
else:
    ...fara exceptie...
```

```
try:
    ...executa cod...
    ...fara exceptie...
except IndexError:
    ...trateaza exceptia...
```

- **Nu este acelasi lucru, pentru ca si codul "fara exceptie" poate declansa exceptia**

Instructiunea...

- Exemplu, comportament implicit (mesaj+terminat):

*# Fisier **bad.py**:*

```
def gobad(x, y):
    return x / y


def gosouth(x):
    print(gobad(x, 0))


gosouth(1)
```

C:\Users\Dan\Desktop>**py -3 bad.py**

```
Traceback (most recent call last):
  File "C:/Users/Dan/Desktop/bad.py", line 7,
    in <module>
    gosouth(1)
  File "C:/Users/Dan/Desktop/bad.py", line 5,
    in gosouth
    print(gobad(x, 0))
  File "C:/Users/Dan/Desktop/bad.py", line 2,
    in gobad
    return x / y
ZeroDivisionError: division by zero
```

Instructiunea...

- Exemplu, interceptarea exceptiilor predefinite:

*# Fisier **kaboom.py**:*

```
def kaboom(x, y):
    print(x + y) # Declanseaza TypeError

try:
    kaboom([0, 1, 2], 'spam') # Concatenare doar a secventelor de acelasi tip!!
except TypeError: # Intercepteaza+tratare exceptie
    print('Hello world!')
print('resuming here') # Continuare, fie exceptie sau nu
```

```
C:\Users\Dan\Desktop>py -3 kaboom.py
Hello world!
resuming here
```

Instructiunea *try/finally*

- Sintaxa:

  try:

  statements # Cod de executat mai intai

  finally:

  statements # Cod executat intotdeauna, la urma

- Executia:
  - Se memoreaza contextul, se executa instructiunile din blocul *try*
  - Daca nu se produce vreo exceptie atunci se executa blocul din clauza *finally* si apoi se continua dupa intreaga instructiune try.
  - Daca se produce o exceptie, intai se executa blocul din clauza *finally* si apoi exceptia este propagata la un *try* anterior sau la tratamentul implicit
- Se foloseste pentru a fi siguri ca o actiune se va produce, indiferent de exceptie

Instructiunea...

- Exemplu, cod final cu *try/finally*:

```
class MyError(Exception): pass

def stuff(file):
    raise MyError()

file = open('data', 'w') # Deschidere fisier pentru scriere
try:
    stuff(file) # Declanseaza o exceptie
finally:
    file.close() # Fisierul este inchis intotdeauna (salvare pe disc)
print('not reached') # Aici se ajunge numai daca nu s-ar fi produs o exceptie
```

  - Desi *garbage collection* inchide fisierele automat in CPython, este mai sigur cu *try/finally*, deoarece alte versiuni de Python (Jython, PyPy) pot avea implementari diferite.

Instructiunea *try/except/finally*

- Incepand cu Python v2.5, clauza *finally* poate apare impreuna cu *except* si *else*. Sintaxa:

```
try: # Combinatie
    main-action
except Exception1:
    handler1
except Exception2: # Interceptare exceptii
    handler2
...
else: # Nici o exceptie
    else-block
finally: # Ultima clauza
    finally-block
```

Instructiunea...

- Executia:
  - Codul din blocul *try* se executa primul
  - Eventualele exceptii se trateaza, iar daca nici o exceptie nu se produce atunci se executa blocul din clauza *else*
  - Blocul din clauza *finally* se executa (chiar daca alta exceptie a fost declansata cu ocazia tratarii exceptiilor).
  - Daca o exceptie este activa, ea este propagata. Altfel se continua dupa intrega instructiune *try*
- Blocul *finally* este executat intotdeauna:
  - Daca o exceptie s-a produs si a fost tratata
  - Daca o exceptie s-a produs si nu a fost tratata
  - Nici o exceptie nu s-a produs
  - O exceptie noua a fost declansata in cursul tratamentului

Instructiunea...

- Sintaxa unificata:
  - *try -> except -> else -> finally*

try: *# Formatul 1*

statements

except *[*type *[*as value*]]*: *# [type [, value]] in Python 2.X*

statements

***[*except *[*type *[*as value*]]*:**

statements]*

***[*else:**

statements]

***[*finally:**

statements]

try: *# Formatul 2*

statements

finally:

statements

- **Macar un *except* este necesar pentru ca *else* sa fie prezent**

Instructiunea...

- Combinarea lui *finally* si *except* prin incluziune:

```
try: # Cod echivalent, cu includere (try in try)
        try:
            main-action
        except Exception1:
            handler1
        except Exception2:
            handler2
        ...
        else:
            no-error
finally:
    cleanup
```

Instructiunea...

- Exemplu, try/except/else/finally:

```
# Fisier mergedexc.py: (Python v3.X + v2.X)
sep = '-' * 45 + '\n'

print(sep + 'EXCEPTION RAISED AND CAUGHT')
try:
    x = 'spam'[99]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')


print(sep + 'NO EXCEPTION RAISED')
try:
    x = 'spam'[3]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')


print(sep + 'NO EXCEPTION RAISED, WITH ELSE')
try:
    x = 'spam'[3]
except IndexError:
    print('except run')
else:
```

Instructiunea...

```
    print('else run')
finally:
    print('finally run')
print('after run')

print(sep + 'EXCEPTION RAISED BUT NOT
    CAUGHT')
try:
    x = 1 / 0
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')
```

- Rezultat:

```
c:\code> py –3 mergedexc.py
----------------------------------------
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
----------------------------------------
NO EXCEPTION RAISED
finally run
after run
----------------------------------------
NO EXCEPTION RAISED, WITH ELSE
else run
finally run
```

Instructiunea...

```
after run                                   Traceback (most recent call last):
-----------------------------------------   File "mergedexc.py", line 39, in <module>
EXCEPTION RAISED BUT NOT CAUGHT             x = 1 / 0
finally run                                 ZeroDivisionError: division by zero
```

Instructiunea *raise*

- Se foloseste la declansarea unei exceptii. Sintaxa:

  raise *instance* # Declanseaza instanta unei clase

  raise *class* # Creeaza o instanta a clasei (fara argumente) si apoi o declanseaza

  raise # Redeclanseaza exceptia cea mai recenta

- Exceptiile sunt intotdeauna instante ale unei clase.
- Exemple:

```
raise IndexError # Clasa (instanta este creata)
raise IndexError() # Instanta (creata pe loc)
exc = IndexError() # Creare instanta in prealabil
raise exc
```

```
excs = [IndexError, TypeError]
raise excs[0]
```

Instructiunea...

- Instanta transmisa de *raise* poate fi asignata unei variabile:

```
try:
    ...
except IndexError as X: # Lui X i se asigneaza
    obiectul instanta
    ...
```

```
class MyExc(Exception): pass
...
raise MyExc('spam') # Instanta creata cu
    argument(e)
...
```

```
try:
    ...
except MyExc as X: # Atributele instantei sunt
    accesibile la tratarea exceptiei
    print(X.args)
```

Instructiunea...

- Domeniul variabilelor din clauza *except*
  - sunt **locale** blocului *except* in Python v3.x (nu in v2.x)

```
>>> try:
        1 / 0
except Exception as X:
    print(X)
division by zero
```

```
>>> X
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    X
NameError: name 'X' is not defined
```

```
>>> X = 99
>>> {X for X in 'spam'} # La fel si in colectii iterative!
{'a', 's', 'p', 'm'}
>>> X
99
```

```
>>> try:
        1 / 0
except Exception as X:
    print(X)
    Saveit = X # Pentru folosire ulterioara
division by zero
>>> X
NameError: name 'X' is not defined
>>> Saveit # Exista!
ZeroDivisionError('division by zero')
```

Instructiunea...

- *raise* poate propaga o exceptie, in formatul fara argument:

```
>>> try:
        raise IndexError('spam')
except IndexError:
        print('propagating')
        raise
propagating
Traceback (most recent call last):
  File "<pyshell#31>", line 2, in <module>
    raise IndexError('spam')
IndexError: spam
```

- Inlantuirea (cauzelor) exceptiilor cu sintaxa ***raise from***:

```
>>> try:
        1 / 0
except Exception as E:
        raise TypeError('Bad') from E
Traceback (most recent call last):
  File "<pyshell#4>", line 2, in <module>
    1 / 0
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<pyshell#4>", line 4, in <module>
    raise TypeError('Bad') from E
TypeError: Bad
```

Instructiunea...

- Inlantuirea (cauzelor) exceptiilor se face automat daca se produce o eroare in interiorul codului care trateaza o exceptie:

```
>>> try:
        1 / 0
except:
        badname
Traceback (most recent call last):
  File "<pyshell#11>", line 2, in <module>
    1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#11>", line 4, in <module>
    badname
NameError: name 'badname' is not defined
```

- Inlantuirile (cauzelor) exceptiilor pot fi oricat de lungi:

```
>>> try:
        try:
            raise IndexError()
        except Exception as E:
            raise TypeError() from E
except Exception as E:
        raise SyntaxError() from E
Traceback (most recent call last):
  File "<pyshell#26>", line 3, in <module>
    raise IndexError()
```

Instructiunea...

```
IndexError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<pyshell#26>", line 5, in <module>
    raise TypeError() from E
TypeError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<pyshell#26>", line 7, in <module>
    raise SyntaxError() from E
  File "<string>", line None
SyntaxError: <no detail available>
```

```
>>> try: # Cauze implicite:
        try:
            1 / 0
        except:
            badname
    except:
        open('nonesuch')
Traceback (most recent call last):
  File "<pyshell#32>", line 3, in <module>
    1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#32>", line 5, in <module>
    badname
```

Instructiunea...

```
NameError: name 'badname' is not defined

During handling of the above exception, another
    exception occurred:

Traceback (most recent call last):
  File "<pyshell#32>", line 7, in <module>
    open('nonesuch')
FileNotFoundError: [Errno 2] No such file or
    directory: 'nonesuch'
```

- Suprimarea afisarii lantului de cauze se poate face cu instructiunea:

  raise newexception from None

Instructiunea *assert*

- Se foloseste la depanarea programelor, reprezentand o instructiune *raise* conditionala. Sintaxa:

assert *test*, *data* # *data* este optional

Echivalent cu:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

  - Cu **python –O main.py** se elimina (optimizeaza) instructiunile *assert* (__debug__ devine *False* din cauza parametrului **-O**)

- Exemplu, detectarea constrangerilor definite de utilizator:

```
>>> def f(x):
        assert x < 0, 'x must be negative'
        return x ** 2
>>> f(1)
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    f(1)
  File "<pyshell#44>", line 2, in f
    assert x < 0, 'x must be negative'
AssertionError: x must be negative
```

Instructiunea...

- *assert* este inutil pentru erori de programare, care sunt detectate automat de catre Python:

```
def reciprocal(x):
    assert x != 0 # Un assert inutil!
    return 1 / x # Python verifica diviziunea la
    zero in mod automat
```

Manageri de context cu *with/as*

- Instructiunea ***with*** impreuna cu clauza optionala ***as*** opereaza asupra obiectelor de tip manager de context – care suporta un protocol cu metode (asemanator cu obiectele iterative)
  - In Python v2.5 este optionala, necesita:
    - **from __future__ import with_statement**
- Sintaxa:

  with *expression* [**as** *variable*]**:**

  with-block
  - *expression* returneaza un obiect ce suporta managementul de context
  - Obiectul poate returna o valoare ce se asigneaza lui *variable* daca clauza ***as*** este prezenta

Manageri...

- Obiectul poate executa un cod initial inainte de executia blocului *with-bloc* si un cod final dupa terminarea blocului, indiferent daca s-a produs sau nu o exceptie.

- Obiecte predefinite in Python, e.g. fisierele, suporta managementul de context – fisierul este inchis automat la sfarsitul blocului:

```
with open(r'C:\misc\data') as myfile:
    for line in myfile:
        print(line)
        ...etc...
# Cod echivalent:
myfile = open(r'C:\misc\data')
try:
    for line in myfile:
        print(line)
        ...etc...
finally:
    myfile.close()
```

Manageri...

- Alt exemplu, firele de executie:

```
import threading
lock = threading.Lock()
with lock:
    # Regiune critica
    ...acces la resurse partajate...
```

- Alt exemplu, modulul ***decimal***:

```
import decimal
with decimal.localcontext() as ctx: # Contextul decimal (precizia, rotunjirea) este restaurat dupa
    blocul lui with
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00')
```

Manageri...

- Protocolul managementului de context:
  - Se bazeaza pe clase cu operatorii predefiniti __*enter*__ si __*exit*__
  - Executia instructiunii *with expression [as variable]*:
    1. Se evalueaza *expression* rezultand un manager de context cu metodele __*enter*__ si __*exit*__
    2. Se apeleaza metoda __*enter*__ iar rezultatul sau este atribuit lui *variable* daca clauza ***as*** este prezenta
    3. Codul din blocul *with-block* este executat
    4. Daca se produce o exceptie, atunci se apeleaza: __*exit*__*(type, value, traceback)* – vezi *sys.exc_info()*
       - daca rezultatul este False, exceptia este redeclansata – normal.
    5. Daca nu se produce nici o exceptie, atunci se apeleaza: __*exit*__*(None, None, None)*

Manageri...

- Exemplu:

```
class TraceBlock:
    def message(self, arg):
        print('running ' + arg)
    def __enter__(self):
        print('starting with block')
        return self # Poate fi si alt obiect
    def __exit__(self, exc_type, exc_value,
exc_tb):
        if exc_type is None:
            print('exited normally\n')
        else:
            print('raise an exception! ' +
str(exc_type))
            return False # Propagare
(oricum None este False)
if __name__ == '__main__': # Autotestare
    with TraceBlock() as action:
        action.message('test 1')
        print('reached')

    with TraceBlock() as action:
        action.message('test 2')
        raise TypeError
        print('not reached')
```

Manageri...

- Rezultat:

```
C:\code>py -3 withas.py
starting with block
running test 1
reached
exited normally


starting with block
running test 2
raise an exception! <class 'TypeError'>
Traceback (most recent call last):
  File "withas.py", line 21, in <module>
    raise TypeError
TypeError
```

Manageri...

- Manageri de context multipli, sintaxa cu virgula:

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        if 'some key' in line:
            fout.write(line)
```

  - Cod echivalent:

```
with A() as a, B() as b:
    ...instructiuni...
```

```
with A() as a:
    with B() as b:
        ...instructiuni...
```

  - Exemplu, scanare in paralel a doua fisiere:

```
>>> with open('script1.py') as f1,
    open('script2.py') as f2:
        for pair in zip(f1, f2):
            print(pair)
('# Un prim script Python\n', 'import sys\n')
('import sys             # Incarca un modul de
    biblioteca, sys denumit.\n',
    'print(sys.path)\n')
('print(sys.platform)\n', 'x = 2\n')
('print(2 ** 100)        # Ridica 2 la putere(a
    32)\n', 'print(x ** 32)\n')
```

Manageri...

- Varianta, cu *enumerate*:

```
with open('script1.py') as f1, open('script2.py') as f2:
    for (linenum, (line1, line2)) in enumerate(zip(f1, f2)):
        if line1 != line2:
            print('%s\n%r\n%r' % (linenum, line1, line2))
```

- Varianta, si mai simpla:

```
for pair in zip(open('script1.py'), open('script2.py')): # Acelasi efect, fisiere inchise automat
    print(pair)
```

- Exemplu, cu continuare dupa eroare, fara *with*:

```
fin = open('script2.py')
fout = open('upper.py', 'w')
try:
    for line in fin:
        fout.write(line.upper())
finally:
    fin.close()
    fout.close()
```