Programarea calculatoarelor si limbaje de programare II

Obiecte de tip exceptie
Proiectarea cu exceptii

Universitatea Politehnica din Bucureşti

Sumar

- ***Avantajele exceptiilor bazate pe clase***
- Obiecte de tip exceptie
- Clase predefinite de exceptii
- Mesaje specializate
- Date si metode ale exceptiilor
- Tratarea exceptiilor incluse
- Tehnici de folosire a exceptiilor
- Recomandari pentru programarea cu exceptii
- Sumar al limbajului Python

Avantaje

- Exceptiile bazate pe clase au urmatoarele avantaje:
  - Pot fi organizate pe **categorii de exceptii** asa incat instructiunile *try* pot ramane nemodificate
  - Au atat **informatii de stare cat si comportament** determinat de metode, utilizate in *try*
  - Suporta **mostenirea** – e.g. a metodelor de afisare a mesajelor de eroare
- Exceptiile predefinite cat si cele definite de utilizator se pot organiza in ierarhii arborescente
- In Python v3.x noile exceptii trebuie sa fie derivate din exceptii predefinite, e.g. *Exception*

Sumar

- Avantajele exceptiilor bazate pe clase
- ***Obiecte de tip exceptie***
- Clase predefinite de exceptii
- Mesaje specializate
- Date si metode ale exceptiilor
- Tratarea exceptiilor incluse
- Tehnici de folosire a exceptiilor
- Recomandari pentru programarea cu exceptii
- Sumar al limbajului Python

Exceptii bazate pe stringuri

- Au fost abandonate incepand cu versiunile v2.6 si v3.0
- Astazi se folosesc numai exceptii bazate pe clase
- Se pot inca intalni in cod mai vechi, e.g.:

```
C:\code> C:\Python25\python
>>> myexc = "My exception string" # Identificarea stringului se face cu operatorul is, nu cu ==
>>> try:
        raise myexc
except myexc:
        print('caught')
caught
```

- Codul actual (v3.7 si v2.7) genereaza exceptii:

```
C:\code>py -3
>>> raise 'spam'
TypeError: exceptions must derive from
      BaseException
```

```
C:\code>py -2
>>> raise 'spam'
TypeError: exceptions must be old-style classes
      or derived from BaseException, not str
```

Exceptii bazate pe clase

- Exceptiile bazate pe *str* sunt testate in clauza *except* cu operatorul *is* – adica cu identitate de obiect
- Exceptiile bazate pe *class* sunt identificate cu relatia dintre clase si superclase – adica orice instanta a unei clase sau superclase poate fi folosita in clauza *except*.
- O superclasa din *except*-ul unui *try* intercepteaza atat instante ale superclasei cat si instante ale subclaselor sale.
  - Superclasa devine o **categorie** de exceptii, organizate **ierarhic**
  - Clasele din ierarhie suporta informatii de stare (in instante) si mostenesc comportamentul – cu metode
- Exceptiile cu *str* ingreunau intretinerea codului, fara organizare pe categorii

Codificarea cu exceptii bazate pe clase

- Superclasa este de obicei implementata cu *pass*, dar se recomanda sa deriveze din exceptia predefinita *Exception*
- Obiectul declansat de *raise* si apoi interceptat in *except* este intotdeauna o instanta a unei clase
  - **raise OExceptie** # *Produce o instanta a clasei chiar si fara paranteze, folosindu-se constructorul fara argumente*
- *try/except* poate intercepta atat categoria de exceptii cat si subclase specifice
- Functia ***sys.exc_info()*** returneaza un tuplu cu informatii despre cea mai recenta exceptie tratata
  - exc_info() -> (type, value, traceback)
  - Practic, *value.__class__* este *type*

Codificarea...

```
# Fisierul classexc.py:
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass
def raiser0():
    X = General() # Instanta a superclasei
    raise X
def raiser1():
    X = Specific1() # Instanta a subclasei
    raise X
def raiser2():
    X = Specific2() # Instanta a altei subclase
    raise X
for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General: # Se intercepteaza fie General, fie oricare subclasa a sa
        import sys
        print('caught: %s'%sys.exc_info()[0])
```

```
C:\code>python classexc.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>
```

- **Codificare alternativa, cu __class__**:

```
try:
    func()
except General as X:
    print('caught: %s' % X.__class__)
```

Rolul ierarhiilor de exceptii

- Codificare alternativa, cu *tuple* de exceptii:

```
try:
    func()
except (General, Specific1, Specific2): # Interceptarea oricarei exceptii din tuplul din clauza except
    ...
```

- Cazul unei biblioteci de functii numerice:

```
# mathlib.py, versiune greu de intretinut
class Divzero(Exception): pass
class Oflow(Exception): pass
def func():
    ...
    raise Divzero()
...etc...
```

```
# client.py
import mathlib
try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow):
    ...tratare erori...
```

Rolul...

- Situatia extinderii bibliotecii cu exceptii noi:

<table>
<tr>
<td>

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

</td>
<td>

```
# client.py, cod dificil de intretinut
try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow,
    mathlib.Uflow):
    ...tratare eroare...
```

</td>
</tr>
<tr>
<td>

```
# client.py, cu clauza except fara argumente
try:
    mathlib.func(...)
except: # Interceptare a oricarei exceptii
    ...tratare eroare...
```

</td>
<td>

- **Problema: se intercepteaza prea multe erori, e.g. memorie insuficienta, Ctrl/C, exit si chiar erori de sintaxa!**
- **Ar trebui interceptate doar erorile specifice bibliotecii**

</td>
</tr>
</table>

- Rezolvarea, cu o ierarhie de exceptii:

Rolul...

```
# mathlib.py
class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
def func():
    ...
    raise DivZero()
...etc...
```

```
# client.py
import mathlib
try:
    mathlib.func(...)
except mathlib.NumErr:
    ...tratare eroare...
```

- La adaugarea unei exceptii noi, codul clientului ramane neschimbat:

```
# mathlib.py
...
class Uflow(NumErr): pass
```

Sumar

- Avantajele exceptiilor bazate pe clase
- Obiecte de tip exceptie
- ***<u>Clase predefinite de exceptii</u>***
- Mesaje specializate
- Date si metode ale exceptiilor
- Tratarea exceptiilor incluse
- Tehnici de folosire a exceptiilor
- Recomandari pentru programarea cu exceptii
- Sumar al limbajului Python

Exceptii predefinite

- Sunt prezente in modulul ***builtins*** si organizate in ierarhii:
  - ***BaseException*** se afla la radacina ierarhiei
    - implementeaza afisarea si un constructor implicit
    - nu se foloseste direct, ci clasa *Exception*
    - afiseaza argumentele constructorului cand __*str*__ este apelat (de *print*)
    - toate argumentele constructorului se memoreaza in atributul ***args*** de tip *tuple*
  - ***Exception*** se afla la radacina exceptiilor definite de utilizator
    - este o subclasa (directa) a lui *BaseException*
    - este superclasa pentru toate exceptiile predefinite, in afara evenimentelor de tip iesire din sistem (*SystemExit*, *KeyboardInterrupt*, *GeneratorExit*)
  - ***ArithmeticError*** se afla la radacina erorilor numerice
    - este subclasa a lui *Exception* si superclasa exceptiilor numerice
    - subclase ale sale: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*

Exceptii...

- ***LookupError*** se afla la radacina erorilor de indexare
  - este subclasa a lui *Exception*
  - este superclasa exceptiilor de indexare a secventelor si a maparilor (*dict*), e.g. *IndexError* si *KeyError*

- In Python v2.x:
  - Exceptiile se afla in modulul ***exceptions***
  - Ierarhia exceptiilor se poate vizualiza cu:

```
>>> import exceptions
>>> help(exceptions)
...etc...
```

Categorii de exceptii, utilizare

- Ierarhizarea exceptiilor permite tratarea exceptiilor intr-un mod mai mult sau mai putin specific:
  - Cu *ArithmeticError* (in *try*) se intercepteaza doar erorile aritmetice
  - Cu *ZeroDivisionError* se intercepteaza doar impartirea la zero
  - Cu *Exception* se intercepteaza toate exceptiile dintr-o aplicatie cu exceptia iesirii din sistem
    - A se evita clauza *except* fara argumente care ar masca toate erorile – chiar si erori de programare/sintaxa

```
try:
    action()
except Exception: # Fara exit aici
    ...tratarea exceptiilor din aplicatie...
else:
    ...cazul nici unei exceptii...
```

Afisarea implicita

- Exceptiile afiseaza in mod implicit atributul de tip *tuple* numit ***args***, creat de constructorul implicit din argumentele sale actuale
  - Un singur argument nu este afisat ca tuplu
  - Cazul exceptiilor predefinite:

```
>>> raise IndexError # Ca IndexError(), fara argumente
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
>>> raise IndexError('spam') # Argumentul constructorului este afisat
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: spam
>>> I = IndexError('spam') # Instanta
>>> I.args # Atribut al instantei
('spam',)
>>> print(I) # Afisarea lui args cu print()
spam
```

Afisarea...

- Cazul exceptiilor definite de utilizator, la fel:

```
>>> class E(Exception): pass
>>> raise E
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise E
E
>>> raise E('spam')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    raise E('spam')
E: spam
>>> I = E('spam')
>>> I.args
('spam',)
>>> print(I)
spam
```

- Acces la argumente si metode:

```
>>> try:
        raise E('spam')
except E as X:
        print(X)
        print(X.args)
        print(repr(X))
spam
('spam',)
E('spam')
```

Afisarea...

- Argumentele multiple se afiseaza ca tuplu:

```
>>> try:
	raise E('spam', 'eggs', 'ham')
except E as X:
	print('%s %s' % (X, X.args))


('spam', 'eggs', 'ham') ('spam', 'eggs', 'ham')
```

Sumar

- Avantajele exceptiilor bazate pe clase
- Obiecte de tip exceptie
- Clase predefinite de exceptii
- ***Mesaje specializate***
- Date si metode ale exceptiilor
- Tratarea exceptiilor incluse
- Tehnici de folosire a exceptiilor
- Recomandari pentru programarea cu exceptii
- Sumar al limbajului Python

Cu __*str*__ sau __*repr*__

- Implicit, se afiseaza argumentele constructorului de instanta
  - la interceptarea si afisarea erorii:

```
>>> class MyBad(Exception): pass
>>> try:
        raise MyBad('Sorry--my mistake!')
except MyBad as X:
        print(X)
Sorry--my mistake!
```

  - sau cand eroarea nu este tratata:

```
>>> raise MyBad('Sorry--my mistake!')
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    raise MyBad('Sorry--my mistake!')
MyBad: Sorry--my mistake!
```

Cu...

- O afisare specializata se poate face cu inlocuirea metodelor __***str***__ sau __***repr***__:

```
>>> class MyBad(Exception):
        def __str__(self):
            return 'Always look on the bright side of life...'
>>> try:
        raise MyBad('Sorry--my mistake!')
except MyBad as X:
        print(X)
Always look on the bright side of life...
```

```
>>> raise MyBad()
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    raise MyBad()
MyBad: Always look on the bright side of life...
```

- Atentie, __*str*__ este de preferat, deoarece exista un __*repr*__ implicit, iar __*str*__ este preferat in locul lui __*repr*__ in context cu *print()* si *str()*

Sumar

- ❑ Avantajele exceptiilor bazate pe clase
- ❑ Obiecte de tip exceptie
- ❑ Clase predefinite de exceptii
- ❑ Mesaje specializate
- ❑ ***Date si metode ale exceptiilor***
- ❑ Tratarea exceptiilor incluse
- ❑ Tehnici de folosire a exceptiilor
- ❑ Recomandari pentru programarea cu exceptii
- ❑ Sumar al limbajului Python

Cu atribute de instanta

- Instanta unei exceptii poate retine atribute ce servesc la tratarea adecvata a erorilor in clauza *try/except*
  - Cu atribute create de constructorul __***init***__:

```
class FormatError(Exception):
    def __init__(self, line, file):
        self.line = line
        self.file = file
def parser():
    raise FormatError(42, file='spam.txt') #
```

Informatii despre eroarea gasita

```
try:
    parser()
except FormatError as X:
    print('Error at: %s %s' % (X.file, X.line))
```

Error at: spam.txt 42

  - Cu ***args*** din superclasa exceptiei (constructor mostenit):

```
class FormatError(Exception): pass
def parser():
    raise FormatError(42, 'spam.txt') # Fara argumente de tip cuvinte cheie
try:
    parser()
except FormatError as X:
    print('Error at:', X.args[0], X.args[1])
```

Error at: 42 spam.txt

Cu metode ale exceptiei

- Metodele definite in clasa exceptiei pot fi apelate la momentul tratarii erorii:

```
# Fisierul excparse.py:
from __future__ import print_function # Pentru v2.X
class FormatError(Exception):
    logfile = 'formaterror.txt'
    def __init__(self, line, file): # Constructor
        self.line = line
        self.file = file
    def logerror(self):  # Metoda care scrie intr-un fisier
        log = open(self.logfile, 'a') # Cu adaugare, la sfarsit
        print('Error at:', self.file, self.line, file=log)

def parser():
    raise FormatError(40, 'spam.txt')
if __name__ == '__main__': # Autotestare
    try:
        parser()
    except FormatError as exc:
        exc.logerror()
```

```
c:\code> del formaterror.txt
c:\code> py -3 excparse.py
c:\code> py -2 excparse.py
c:\code> type formaterror.txt
Error at: spam.txt 40
Error at: spam.txt 40
```

Cu...

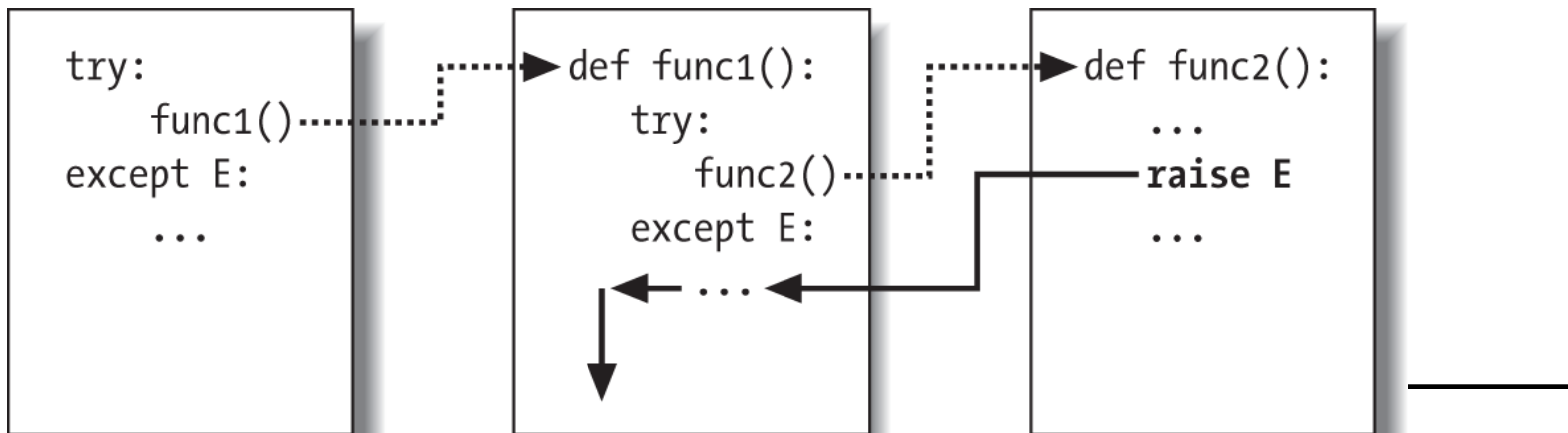
- In cazul unei clauze *except* fara argumente, obiectul instanta a celei mai recente erori este accesibil cu apelul (*import sys* mai intai): **sys.exc_info()[1]**

Sumar

- ❑ Avantajele exceptiilor bazate pe clase
- ❑ Obiecte de tip exceptie
- ❑ Clase predefinite de exceptii
- ❑ Mesaje specializate
- ❑ Date si metode ale exceptiilor
- ❑ ***Tratarea exceptiilor incluse***
- ❑ Tehnici de folosire a exceptiilor
- ❑ Recomandari pentru programarea cu exceptii
- ❑ Sumar al limbajului Python

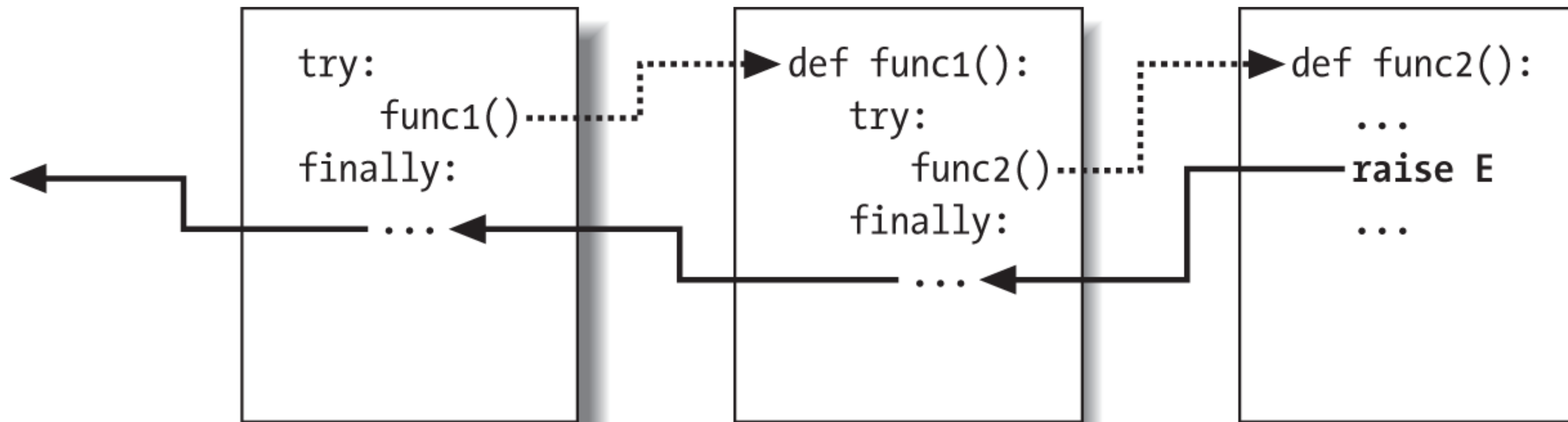
Tratarea erorilor cu instructiuni ***try*** incluse

- Instructiunile *try* pot fi incluse, atat **sintactic** dar si din punct de vedere al **ordinii de executie**
  - La executie, toate instructiunile *try* incepute dar nefinalizate sunt memorate cu o structura de tip stiva
  - Cand se produce o eroare, controlul revine celei mai recente clauze *except* care poate intercepta respectiva eroare.
    - executia continua dupa blocul *try* care a tratat eroarea:



Tratarea...

- Clauzele *finally* sunt toate executate:



- Exemplu de incluziune la nivel de executie:

```
def action2():
    print(1 + []) # Se produce TypeError
def action1():
    try:
        action2()
    except TypeError: # Clauza cea mai recenta
        print('inner try')
try:
    action1()
except TypeError:
    print('outer try')
```

- **Rezultat:** inner try

Tratarea...

- Exemplu de incluziune sintactica – dar cu acelasi efect:

```
try:
    try:
        action2()
    except TypeError: # Clauza cea mai recenta
        print('inner try')
except TypeError:
    print('outer try')
```

- **Rezultat:** inner try

- Exemplu cu clauze *finally* (toate executate):

```
>>> try:
        try:
            raise IndexError
        finally:
            print('spam')
finally:
        print('SPAM')
```

```
spam
SPAM
Traceback (most recent call last):
  File "<pyshell#7>", line 3, in <module>
    raise IndexError
IndexError
```

Tratarea...

- Exemplu cu *except* si *finally* (fisier *except-finally.py*):

```
def raise1(): raise IndexError
def noraise(): return
def raise2(): raise SyntaxError # Se propaga
for func in (raise1, noraise, raise2):
    print('<%s>' % func.__name__)
    try:
        try:
            func()
        except IndexError:
            print('caught IndexError')
    finally:
        print('finally run')
    print('...')
```

```
<raise1>
caught IndexError
finally run
...
<noraise>
finally run
...
<raise2>
finally run
Traceback (most recent call last):
File "except-finally.py", line 9, in <module>
func()
File "except-finally.py", line 3, in raise2
def raise2(): raise SyntaxError
SyntaxError: None
```

Sumar

- ❑ Avantajele exceptiilor bazate pe clase
- ❑ Obiecte de tip exceptie
- ❑ Clase predefinite de exceptii
- ❑ Mesaje specializate
- ❑ Date si metode ale exceptiilor
- ❑ Tratarea exceptiilor incluse
- ❑ ***Tehnici de folosire a exceptiilor***
- ❑ Recomandari pentru programarea cu exceptii
- ❑ Sumar al limbajului Python

Salt in afara oricator cicluri

- Instructiunea ***break*** permite iesirea doar din ciclul curent, dar ***raise*** si ***except*** permit saltul de tip *goto*:

```
>>> class Exitloop(Exception): pass
>>> try:
        while True:
            while True:
                for i in range(10):
                    if i > 3: raise Exitloop # goto
                    print('loop3: %s' % i)
                print('loop2')
            print('loop1')
except Exitloop:
    print('continuing') # Continuare, sau chiar pass
```

```
loop3: 0
loop3: 1
loop3: 2
loop3: 3
continuing
>>> i # Variabilele din try sunt pastrate
4
```

Exceptii care nu sunt erori

- Exceptia ***EOFError*** semnalizeaza sfarsitul de fisier de intrare, e.g. *sys.stdin*, in cazul apelurilor successive ale functiei predefinite ***input()***:

```
while True:
    try:
        line = input() # Citirea liniei urmatoare (raw_input in v2.X)
    except EOFError:
        break # Iesire normala, la sfarsit de fisier
    else:
        ...procesarea liniei curente...
```

- Exceptiile ***SystemExit*** si ***KeyboardInterrupt*** semnalizeaza executia lui ***sys.exit()*** si introducerea combinatiei ***Ctrl/C*** de la tastatura, respectiv

Exceptii...

- Exceptii care reprezinta simple avertizari, nu erori, e.g. pentru elemente ale limbajului Python ce urmeaza sa dispara, se afla in modulul ***warnings***:

```
>>> import warnings
>>> help(warnings)
```

Exceptii definite de utilizator care nu sunt erori

- Functiile pot semnaliza exceptii definite de programator in locul returnarii unui rezultat specific, e.g. la gasirea unui element intr-o functie de cautare, deci cu *try/except/else* in loc de tehnica testarii rezultatului cu *if/else*:

```
class Found(Exception): pass
def searcher():
    if ...succes...:
        raise Found() # Semnalizare exceptie
    else: # Cautare esuata
        return
```

```
try:
    searcher()
except Found: # Cautare reusita!
    ...succes...
else: # Nici o exceptie, deci cautare esuata!
    ...esuare...
```

Exceptii...

- Aceeasi tehnica este obligatorie daca orice (obiect) rezultat este o valoare posibila:

```
class Failure(Exception): pass
def searcher():
    if ...succes...:
        return ...obiect gasit...
    else:
        raise Failure()
```

```
try:
    item = searcher()
except Failure:
    ...negasit...
else:
    ...folosirea lui item...
```

Inchiderea fisierelor si a conexiunilor

- Serverele din retea trebuie sa inchida conexiunile cu clientii care incheie o sesiune, fara ca procesul server sa se termine (asteapta alti clienti)
- Fisierele de iesire trebuie inchise pentru a se salva continutul pe disc
- Tehnici folosite:
  - Cu instructiunile *try/finally* – explicit, mai general:

```
myfile = open(r'C:\code\textdata', 'w')
try:
    ...prelucrare myfile...
finally:
    myfile.close()
```

Inchiderea...

- Cu manageri de context – implicit, mai putin general, mai putin cod, suporta si actiuni initiale:

```
with open(r'C:\code\textdata', 'w') as myfile:
    ...prelucrare myfile...
```

Depanarea programelor cu *try*

- Tot scriptul poate fi incadrat intr-o instructiune *try/except*:

```
try:
    ...executie program...
except: # Aici se trateaza toate exceptiile
    neinterceptate in program
    import sys
    print('uncaught!', sys.exc_info()[0],
    sys.exc_info()[1]) # In pozitia zero este
    clasa, iar in pozitia 1 este instanta
```

Depanarea modulelor importate

- Depanarea mai multor programe, importate, se poate face continuind dupa erori care altfel ar termina modulul curent:

```
import sys
log = open('testlog', 'a')
from testapi import moreTests, runNextTest, testName

def testdriver():
    while moreTests():
        try:
            runNextTest()
        except:
            print('FAILED', testName(), sys.exc_info()[:2], file=log)
        else:
            print('PASSED', testName(), file=log)

testdriver()
```

Functia *sys.exc_info()*

- Este utila la accesarea generala a exceptiilor, e.g. intr-o clauza *except* fara argumente:

```
try:
    ...
except:
    # sys.exc_info()[0:2] sunt clasa si instanta exceptiei
```

- Returneaza un *tuple* – (*type*, *value*, *traceback*):
  - *type* este clasa exceptiei curent tratata
  - *value* este instanta exceptiei declansate
  - *traceback* este un obiect care reprezinta stiva apelurilor – vezi modulul ***traceback***

Modulul *traceback*

- Poate fi folosit la afisarea stivei de executie si a mesajului de eroare pentru o exceptie, manual:

```
import traceback
def inverse(x):
    return 1 / x
try:
    inverse(0)
except Exception:
    traceback.print_exc(file=open('badly.exc', 'w'))
print('Bye')
```

- **Rezultat:**

```
>>> with open(r'badly.exc') as f:
        print(f.read())
Traceback (most recent call last):
  File "C:/Users/Dan/Desktop/CURS-PYTHON/test.py", line 6, in <module>
    inverse(0)
  File "C:/Users/Dan/Desktop/CURS-PYTHON/test.py", line 4, in inverse
    return 1 / x
ZeroDivisionError: division by zero
```

Sumar

- Avantajele exceptiilor bazate pe clase
- Obiecte de tip exceptie
- Clase predefinite de exceptii
- Mesaje specializate
- Date si metode ale exceptiilor
- Tratarea exceptiilor incluse
- Tehnici de folosire a exceptiilor
- ***Recomandari pentru programarea cu exceptii***
- Sumar al limbajului Python

Reguli de folosire a instructiunii *try*

- Se recomanda folosirea lui *try* pentru:
  - **Operatii care pot esua**: interfata cu sistemul de operare – deschidere de fisiere, de conexiuni in retea, etc.
  - Se excepteaza situatiile in care erorile se intentioneaza a fi interceptate de Python
  - Efectuarea de operatii **finale**: cu *try/finally* sau *with/as*
  - La **apelul** unei functii, in loc de clauze *try/except* multiple, pe parcursul functiei
- Programele de tip server folosesc *try* pentru a nu-si incheia executia in caz de erori
- Testarea programelor externe necesita de asemenea *try*.

Evitarea interceptarilor excesive

- Cu un *except* fara argumente se pot intercepta exceptii in mod prematur:

```
def func():
    try:
        ... # Aici se produce IndexError
    except:
        ... # Dar este interceptata aici, prematur!

try:
    func()
except IndexError: # Exceptia trebuie tratata aici!!
    ...
```

- Impiedicarea terminarii unui script cu apelul ***sys.exit(statuscode)***:

```
import sys
def bye():
    sys.exit(40) # Abortare acum!
try:
    bye()
except:
    print('got it') # Exit-ul este ignorat...
print('continuing...')
got it
continuing...
```

Evitarea...

- Corect:

```
try:
    bye()
except Exception: # Nu intercepteaza exit, dar foarte multe exceptii...
    ...
```

- Si totusi, se vor intercepta, in mod nedorit, chiar si erorile de sintaxa/scriere:

```
mydictionary = {...}
...
try:
    x = myditctionary['spam'] # Eroare de scriere, va produce SyntaxError
except:
    x = None # Se presupune ca a fost un KeyError, nu SyntaxError...
...continuare cu x...
```

- Corect, specific: ***except KeyError:***

Interceptari insuficiente

- Numai exceptiile listate – *tuple* cu paranteze – in *except* vor fi interceptate:

try:

...

except (MyExcept1, MyExcept2): *# Codul va fi eronat daca devine posibil un MyExcept3*

... *# Caz fara erori*

else:

... *# Caz de eroare*

- **Corect, cu o superclasa:**

try:

...

except SuccessCategoryName: *# OK*

... *# Caz fara erori*

else:

... *# Caz de eroare*

Sumar

- ❑ Avantajele exceptiilor bazate pe clase
- ❑ Obiecte de tip exceptie
- ❑ Clase predefinite de exceptii
- ❑ Mesaje specializate
- ❑ Date si metode ale exceptiilor
- ❑ Tratarea exceptiilor incluse
- ❑ Tehnici de folosire a exceptiilor
- ❑ Recomandari pentru programarea cu exceptii
- ❑ ***Sumar al limbajului Python***

Ierarhia instrumentelor Python

- Instrumente Python:
  - **Obiecte predefinite:** *str*, *list*, *dict*, *set*, *tuple*, etc.
  - **Extensii Python**: functii, clase, module scrise de programator
  - **Extensii compilate**: module scrise in alte limbaje, C, C++

Suport pentru dezvoltari ample in Python

- Programele ample se scriu mai usor cu:
  - **PyDoc** si **docstrings** – pentru documentarea codului
  - **PyChecker** si **PyLint** – pentru gasirea erorilor in cod
  - **PyUnit** (***unittest***) – sistem de testare a programelor
  - **doctest** – modul standard de test
  - **IDE** pentru Python – Eclipse, Komodo, NetBeans
  - **Optimizari de cod** – cu *timeit* sau cu modulul ***profile***, care raporteaza statistici de executie:
    - *profile.run('cod')* sau *python -m profile args*
    - sau cu modulul ***Cprofile*** - mai simplu
  - **Depanari** – cu ***pdb***
    - *pdb.run('main()')* sau *python -m pdb main.py*
  - **Pachete Python** – cu *py2exe*, *PyInstaller* care produc *frozen binary* sau cu modulul **distutils**

Suport...

- **Optimizari de cod** cu:
  - **PyPy** (JIT)
  - optiunea **-O** in linia de comanda python (rezulta fisiere cu extensia **.pyo**)
  - **Shed Skin** care translateaza codul Python in C++
  - scrierea unor portiuni de cod in C
- Vezi http://www.python.org si alte resurse Web!