Programarea calculatoarelor si limbaje de programare II

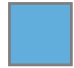
Module: notiuni avansate

Universitatea Politehnica din Bucureşti

Sumar

- ***<u>Concepte in proiectarea modulelor</u>***
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __*future*__
- Utilizare mixta cu __*name*__ si __*main*__
- Exemplu de cod dual
- Modificarea lui *sys.path*
- Extensia *as* pentru *import* si *from*
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

Recomandari pentru proiectarea modulelor



- Codul Python este intotdeauna scris intr-un modul
  - sesiunea interactiva (e.g. Idle) este in modulul predefinit __**main**__ – unde expresiile sunt afisate automat si codul nu este salvat pe disc (se pierde)
- Interactiunea intre module trebuie minimizata
  - variabile globale nu trebuie sa fie importate, cu exceptia functiilor si claselor
- Coeziunea modulelor – interactiune redusa
  - se obtine grupand componente destinate rezolvarii unui singur obiectiv
- Modulele nu trebuie sa modifice direct variabilele altor module
  - rezultatele se transmit prin valorile returnate de functii cu o interfata bine definita

Recomandari...

- Mediul de executie al modulelor:

Modules
Variables
Functions
Variables
Classes
Variables
Methods
Variables

import

Other modules
(Python or C)
Variables
Functions
Classes/Types

import

Other modules
(Python or C)
Variables
Functions
Classes/Types

Sumar

- Concepte in proiectarea modulelor
- ***Mascarea datelor din module***
- Caracteristici viitoare cu modulul __*future*__
- Utilizare mixta cu __*name*__ si __*main*__
- Exemplu de cod dual
- Modificarea lui *sys.path*
- Extensia *as* pentru *import* si *from*
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

import * cu _X si __all__

- Modulele exporta toate numele asignate in afara functiilor si claselor
- Ascunderea datelor din module se face prin conventie, iara nu prin constrangeri sintactice
- Variabile prefixate cu un singur underscore, e.g. _**X**, nu vor fi importate cu instructiunea ***import* ***
  - astfel se evita importarea prea multor variabile
  - importarea fara * permite accesul la orice variabila

```
# Fisierul unders.py:
a, _b, c, _d = 1, 2, 3, 4
>>> from unders import * # Se incarca numele
    care nu sunt de tipul _X
>>> a, c
(1, 3)
>>> _b
NameError: name '_b' is not defined
>>> import unders # Acces si la nume de tipul _X
>>> unders._b
2
```

from...

- Plasand la inceptul modulului o variabila de tip *list*, numita __**all**__, initializata cu nume de variabile din modul – **ca *str***, acestea vor fi singurele importate de *import* * (chiar daca sunt de tipul _X)
  - __all__ din fisierele __init__.py de initializare a pachetelor precizeaza care submodule sunt incarcate cu un *import* * din modulul care le cuprinde

```
# Fisierul alls.py:
__all__ = ['a', '_c'] # __all__ are precedenta fata de _X
a, b, _c, _d = 1, 2, 3, 4
>>> from alls import * # Doar variabilele din __all__ se incarca
>>> a, _c
(1, 3)
>>> b
NameError: name 'b' is not defined
>>> from alls import a, b, _c, _d # Acces la toate
>>> a, b, _c, _d
(1, 2, 3, 4)
>>> import alls
>>> alls.a, alls.b, alls._c, alls._d
(1, 2, 3, 4)
```

Sumar

- ❑ Concepte in proiectarea modulelor
- ❑ Mascarea datelor din module
- ❑ ***Caracteristici viitoare cu modulul __future__***
- ❑ Utilizare mixta cu __*name*__ si __*main*__
- ❑ Exemplu de cod dual
- ❑ Modificarea lui *sys.path*
- ❑ Extensia *as* pentru *import* si *from*
- ❑ Exemplu, modulul ca obiect in Python
- ❑ Importarea cu nume ca string
- ❑ Reincarcarea tranzitiva a modulelor
- ❑ Erori in folosirea modulelor

Modulul __*future*__

- Imbunatatiri ale limbajului sunt introduse treptat, sub forma unor extensii optionale, implicit dezactivate
- Activarea se face cu urmatorul *import*:

from __future__ import *featurename*

- Aceasta instructiune trebuie sa apara prima (dupa docstrings) in script sau intr-o sesiune interactiva
- Exemple, din v3.x spre uz in v2.x:

```
>>> from __future__ import division
>>> from __future__ import print_function
>>> from __future__ import absolute_import
```

- Instructiunea *from __future__...* poate ramane, chiar pentru versiuni care suporta deja respectiva imbunatatire

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- ***Utilizare mixta cu __name__ si __main__***
- Exemplu de cod dual
- Modificarea lui *sys.path*
- Extensia *as* pentru *import* si *from*
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

__name__ si "__main__"

- Un modul poate fi atat importat cat si executat ca un script, prin testarea atributului implicit __**name**__:
  - Daca fisierul este rulat ca script, __*name*__ va fi egal cu **"__main__"**
  - Daca este importat, atunci __*name*__ va fi numele cu care a fost importat

```
# Fisier runme.py:
def tester():
    print("Ce-ti doresc eu tie...")
if __name__ == '__main__': # Este script?
    tester()
C:\code> python
```

```
>>> import runme # Importare
>>> runme.tester() # Apel functie
Ce-ti doresc eu tie...
C:\code> python runme.py # Ca script, tester()
    se executa automat
Ce-ti doresc eu tie...
```

__name__...

- Tehnica serveste la:
  - includerea unui cod de testare a modulului, la sfarsit, cu testul **__name__ == '__main__'** si executia ca script principal
  - scrierea de module de biblioteca care pot fi folosite si ca utilitare pentru linia de comanda
- Modulul **minmax2.py**:

```
print('I am:', __name__)
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res
def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y
if __name__ == '__main__': # Cod pentru autotestare
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

__name__...

- <u>Executia ca script</u>: <u>Prin importare</u>:

Executia ca script	Prin importare
C:\code>**python minmax2.py**	C:\code>python
I am: **__main__**	>>> **import minmax2**
1	I am: **minmax2**
6	>>> **minmax2.minmax(minmax2.lessthan, 's', 'p', 'a', 'm')**
	'a'

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- Utilizare mixta cu __name__ si __main__
- ***Exemplu de cod dual***
- Modificarea lui *sys.path*
- Extensia *as* pentru *import* si *from*
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

formats.py

- Modulul *formats.py*:
  - defineste doua functii, *commas()* si *money()*
  - daca ruleaza ca script, include cod de autotestare, in functia *selftest()*
    - ia in considerare doua argumente din linia de comanda, daca prezente, pentru a efectua un test particularizat – cu lista **sys.argv**
  - include docstrings! Ca in tema de casa ☺

```
#!python
"""
Fisier: formats.py (2.X si 3.X)
Afisare de stringuri numerice.
Se poate autotesta sau cu argumentele din linia
de comanda
"""
```

```
def commas(N):
    """
    Afiseaza intregul pozitiv N cu virgule dupa
        grupuri de trei cifre: "12,345,678".
    """
    digits = str(N)
    ...
```

formats...

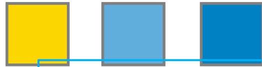
```
    assert digits.isdigit()
    result = ''
    while digits:
        digits, last3 = digits[:-3], digits[-3:]
        result = (last3 + ',' + result) if result
        else last3
    return result
def mycommas(N): # Versiune ca in C!!
```

```
dig = str(N)
assert dig.isdigit(), 'Only digits!'
rst = len(dig) % 3
res = dig[:rst]
for i in range(0, len(dig) // 3 * 3, 3):
    res += (',' if res else '') + dig[i + rst:i +
rst + 3]
return res
```

```
def money(N, numwidth=0, currency='$'):
    """
N afisat cu virgule, 2 zecimale, $, semn si
    optional spatii: "$ -12,345.78".
numwidth=0 fara spatii, currency='' ca sa fie
omisa si non-ASCII pentru alte valute (e.g.,
GBP =u'\xA3' sau u'\u00A3').
    """
```

```
sign = '-' if N < 0 else ''
N = abs(N)
whole = commas(int(N))
fract = ('%.2f' % N)[-2:] # Doar ultimile doua
number = '%s%s.%s' % (sign, whole, fract)
return '%s%*s' % (currency, numwidth,
number)
```

formats...



```
if __name__ == '__main__': # Ca script
    def selftest():
        tests = 0, 1 # rateaza cu: −1, 1.23
        tests += 12, 123, 1234, 12345,
    123456, 1234567
        tests += 2 ** 32, 2 ** 100
        for test in tests:
            print(commas(test))
        print()
        tests = 0, 1, −1, 1.23, 1., 1.2, 3.14159
        tests += 12.34, 12.344, 12.345,
    12.346
        tests += 2 ** 32, (2 ** 32 + .2345)
        tests += 1.2345, 1.2, 0.2345
        tests += −1.2345, −1.2, −0.2345
        tests += −(2 ** 32), −(2**32 + .2345)
        tests += (2 ** 100), −(2 ** 100)
        for test in tests:
            print('%s [%s]' % (money(test,
    17), test))

    import sys # sys.argv lista argumentelor
    if len(sys.argv) == 1: # Fara argumente
        selftest()
    else: # argv[0] este numele programului
        print(money(float(sys.argv[1]),
    int(sys.argv[2])))
```

formats...

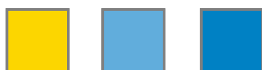
- Testare ca script:

```
C:\code>python formats.py
0
1
12
123
1,234
12,345
...etc...
C:\code>python formats.py 1234 0
$1,234.00
C:\code>python formats.py -1234 0
$-1,234.00
C:\code>python formats.py 123456789012345 0
$123,456,789,012,345.00
C:\code>python formats.py -123456789012345 25
$  -123,456,789,012,345.00
```

- Testare prin importare:

```
>>> from formats import money, commas
>>> money(123.456)
'$123.46'
>>> money(-9999999.99, 15)
'$ –9,999,999.99'
>>> X = 99999999999
>>> '%s (%s)' % (commas(X), X)
'99,999,999,999 (99999999999)'
```

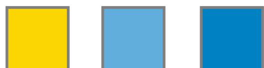
Valute, cu Unicode



- Pentru alte simboluri valutare, reprezentare cu Unicode:
  - ***str*** precedat de **u**, secvente escape in hexagesimal
  - ***bytes***, string precedat de **b**, decodat

```
C:\code>py -3
>>> from __future__ import print_function # Eventual v2.X
>>> from formats import money
>>> X = 54321.987
>>> print(money(X), money(X, 0, '$'))
$54,321.99 54,321.99
>>> print(money(X, currency=u'\xA3'), money(X, currency=u'\u00A5'))
£54,321.99 ¥54,321.99
>>> print(money(X, currency=b'\xA3'.decode('latin-1')))
£54,321.99
>>> print(money(X, currency=u'\u20AC'), money(X, 0, b'\xA4'.decode('iso-8859-15')))
€54,321.99 €54,321.99
>>> print(money(X, currency=b'\xA4'.decode('latin-1’))) #valuta nespecificata
¤54,321.99
```

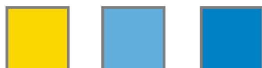
Valute...



- In v2.x sunt necesare ajustari de code page, PYTHONIOENCODING:

```
C:\code>chcp 65001
Active code page: 65001
C:\code>set PYTHONIOENCODING=utf-8
C:\code>py -2 formats_currency.py > temp
C:\Users\Dan\Desktop\code>type temp
$54,321.99 54,321.99
£54,321.99 ¥54,321.99
£54,321.99
€54,321.99 €54,321.99
¤54,321.99
```

Documentare cu docstrings



- Inspectarea documentatiei prezenta sub forma de docstringuri se face cu functia **help()** in mod text si cu **PyDoc** in mod grafic:

```
C:\code>python
>>> import formats
>>> help(formats)
Help on module formats:

NAME
    formats

DESCRIPTION
    File: formats.py (2.X and 3.X)
    Various specialized string display formatting
      utilities.
    Test me with canned self-test or command-
      line arguments.
    To do: add parens for negative money, add
      more features.

FUNCTIONS
    commas(N)
        Format positive integer-like N for display
        with
        commas between digit groupings:
        "xxx,yyy,zzz".
...etc...
```

Documentare...

```
C:\code>python -m pydoc -b
Server ready at http://localhost:63480/
Server commands: [b]rowser, [q]uit
q
Server stopped
```

Pydoc: module formats | Settings

localhost:63480/formats.html

Python 3.7.4 [tags/v3.7.4:e09359112e, MSC v.1916 64 bit (AMD64)]
Windows-10

Module Index : Topics : Keywords

Get | Search

formats

index
c:\users\dan\desktop\code\formats.py

```
File: formats.py (2.X and 3.X)
Various specialized string display formatting utilities.
Test me with canned self-test or command-line arguments.
To do: add parens for negative money, add more features.
```

Functions

commas(N)

```
    Format positive integer-like N for display with
    commas between digit groupings: "xxx,yyy,zzz".
```

money(N, numwidth=0, currency='$')

```
    Format number N for display with commas, 2 decimal digits,
    leading $ and sign, and optional padding: "$  -xxx,yyy.zz".
    numwidth=0 for no space padding, currency='' to omit symbol,
    and non-ASCII for others (e.g., pound=u'£' or u'£').
```

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- Utilizare mixta cu __name__ si __main__
- Exemplu de cod dual
- ***Modificarea lui sys.path***
- Extensia *as* pentru *import* si *from*
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

Modificarea lui *sys.path*

- In afara de **PYTHONPATH** si fisiere cu extensia **.pth**, calea de cautare a modulelor se poate schimba prin alterarea listei **sys.path**, in mod arbitrar:

```
C:\code>python
>>> import sys
>>> sys.path
['', 'C:\\Program Files\\Python37\\python37.zip',
    'C:\\Program Files\\Python37\\DLLs',
    'C:\\Program Files\\Python37\\lib',
    'C:\\Program Files\\Python37', 'C:\\Program
    Files\\Python37\\lib\\site-packages']
>>> sys.path = [r'd:\temp'] # Exista doar o
    singura copie a modulului sys in memorie
>>> sys.path.append('c:\\lp5e\\examples') #
    Valabil doar in aceasta sesiune
>>> sys.path.insert(0, '..')
>>> sys.path
['..', 'd:\\temp', 'c:\\lp5e\\examples']
>>> import string
ImportError: No module named 'string'
```

- Schimbarile lui *sys.path* sunt dinamice (per proces), in timp ce PYTHONPATH (per utilizator) si **.pth* (per instalatie) sunt permanente.

Sumar

- ❑ Concepte in proiectarea modulelor
- ❑ Mascarea datelor din module
- ❑ Caracteristici viitoare cu modulul __future__
- ❑ Utilizare mixta cu __name__ si __main__
- ❑ Exemplu de cod dual
- ❑ Modificarea lui sys.path
- ❑ ***Extensia as pentru import si from***
- ❑ Exemplu, modulul ca obiect in Python
- ❑ Importarea cu nume ca string
- ❑ Reincarcarea tranzitiva a modulelor
- ❑ Erori in folosirea modulelor

as

- Instructiunile *import* si *from* pot avea extensia **as**:

import modulename as name *# Mai departe se foloseste name, nu modulename*	**import modulename** *# Cod echivalent* **name = modulename** **del modulename** *# Se sterge*
from modulename import attrname as name *# Mai departe se foloseste name, nu attrname*	

- **as** este util pentru sinonime mai scurte sau pentru evitarea conflictelor de nume deja utilizate:

import reallylongmodulename as name *# name este sinonim scurt* **name.func()**	**from module1 import utility as util1** *# Ambele utility se pot folosi ca util1 si util2* **from module2 import utility as util2** **util1(); util2()**

as...

- Caile extinse de module pot fi prescurtate la importare:

```
import dir1.dir2.mod as mod # Calea completa este mentionata o singura data
mod.func()
```

- Numele importate pot fi facute unice, spre a se evita conflictele de nume:

```
from dir1.dir2.mod import func as modfunc # Redenumire unica
modfunc()
```

- In cazul unor biblioteci cu nume noi se poate folosi temporar **as** cu numele vechi:

```
import newname as oldname
from library import newname as oldname
# oldname se poate folosi in continuare pana la actualizarea codului
# e.g. v3.x tkinter vs. v2.x Tkinter
```

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- Utilizare mixta cu __name__ si __main__
- Exemplu de cod dual
- Modificarea lui sys.path
- Extensia *as* pentru import si from
- ***Exemplu, modulul ca obiect in Python***
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

Obiectul modul

- Atributele obiectului de tip modul pot fi folosite la crearea unor metaprograme ce realizeaza inspectia altor obiecte, a componentelor acestora
- Fie **M** un modul cu atributul **name** – poate fi accesat in diverse moduri:

M.name # *Calificare obisnuita* **M.__dict__['name']** # Indexare dupa str-ul 'name' in atributul implicit de tip dict __dict__	**sys.modules['M'].name** # *Indexare dupa numele modulului* 'M' in dict-ul sys.modules=>obiect modul + calificare **getattr(M, 'name')** # *Apel al functiei predefinite **getattr** cu arg. obiect modul si str nume de atribut*
• Intr-o functie: **global x; x = 0** este echivalent cu: **import sys; gl=sys.modules[__name__]; gl.x =**	**0** deoarece functia vede atributul implicit **__name__**

Modulul *mydir.py*

- Exporta functia *listing()*, ca functia predefinita dir():

```
#!python
"""
mydir.py: modul care listeaza atributele unui
    modul
"""
from __future__ import print_function # Pentru
    compatibilitate cu v2.X
seplen = 60
sepchr = '-'
def listing(module, verbose=True):
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print('name:', module.__name__,
'file:', module.__file__) # Atr. __file__!
        print(sepline)
count = 0
for attr in sorted(module.__dict__): #
Iterare pe cheile dictionarului
        print('%02d) %s' % (count, attr), end
= ' ')
        if attr.startswith('__'):
            print('<built-in name>') # Nume
predefinite
        else:
            print(getattr(module, attr)) #
Ca module .__dict__[attr]
        count += 1
```

Modulul...

```
if verbose:
    print(sepline)
    print(module.__name__, 'has %d names' % count)
    print(sepline)

if __name__ == '__main__':
    import mydir
    listing(mydir) # Autotestare ca script principal
```

- Executie, ca script:

```
C:\code>py -3 mydir.py
------------------------------------------------------------
name: mydir file: C:\code\mydir.py
------------------------------------------------------------
00) __builtins__ <built-in name>
01) __cached__ <built-in name>
02) __doc__ <built-in name>
03) __file__ <built-in name>
...etc...
05) __name__ <built-in name>
08) listing <function listing at
    0x00000149F565A558>
09) print_function _Feature((2, 6, 0, 'alpha', 2),
    (3, 0, 0, 'alpha', 0), 65536)
10) sepchr -
11) seplen 60
------------------------------------------------------------
mydir has 12 names
------------------------------------------------------------
```

Modulul...

- Executie, ca modul:

```
C:\code>py -3
>>> import mydir
>>> import tkinter
>>> mydir.listing(tkinter)
-------------------------------------------------------
name: tkinter file: C:\Program
      Files\Python37\lib\tkinter\__init__.py
-------------------------------------------------------
00) ACTIVE active
01) ALL all
02) ANCHOR anchor
03) ARC arc
04) BASELINE baseline
...etc...
153) getint <class 'int'>
154) image_names <function image_names at
      0x0000029F461A23A8>
155) image_types <function image_types at
      0x0000029F46202438>
156) mainloop <function mainloop at
      0x0000029F461E2828>
157) re <module 're' from 'C:\\Program
      Files\\Python37\\lib\\re.py'>
158) sys <module 'sys' (built-in)>
159) wantobjects 1
-------------------------------------------------------
tkinter has 160 names
-------------------------------------------------------
```

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- Utilizare mixta cu __name__ si __main__
- Exemplu de cod dual
- Modificarea lui sys.path
- Extensia *as* pentru import si from
- Exemplu, modulul ca obiect in Python
- ***Importarea cu nume ca string***
- Reincarcarea tranzitiva a modulelor
- Erori in folosirea modulelor

Importare dintr-un *str* ca nume de modul

- <u>Sintaxa eronata de importare</u>:

```
>>> import 'string'
SyntaxError: invalid syntax
```

```
>>> x = 'string'
>>> import x
ModuleNotFoundError: No module named 'x'
```

- Cu functia predefinita **exec()** se poate executa codul dintr-un string – ce contine un import
  - pentru expresii exista **eval()**

```
>>> modname = 'string'
>>> exec( 'import  ' + modname ) # Executia codului din argumentul de tip str
```

```
>>> string # Importat aici
<module 'string' from 'C:\\Program Files\\Python37\\lib\\string.py'>
```

- <u>Cu functia predefinita ***__import__()***</u>:

```
>>> modname = 'string'
>>> string = __import__(modname)
>>> string
<module 'string' from 'C:\\Program Files\\Python37\\lib\\string.py'>
```

Importare...

- Deoarece *__import__()* este folosita de interpretorul de Python, se recomanda pentru programare folosirea functiei ***importlib.import_module()***:

```
>>> import importlib
>>> modname = 'string'
>>> string = importlib.import_module(modname)
>>> string
<module 'string' from 'C:\\Program Files\\Python37\\lib\\string.py'>
```

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- Utilizare mixta cu __name__ si __main__
- Exemplu de cod dual
- Modificarea lui sys.path
- Extensia *as* pentru import si from
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- ***Reincarcarea tranzitiva a modulelor***
- Erori in folosirea modulelor

reloadall.py

- Functia *importlib.reload()* incarca doar un singur modul – deja importat, chiar daca modulul, la randul sau, efectueaza si alte importari:

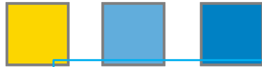
```
# Fisierul A.py:
import B # B, C nu sunt reincarcate!
import C # Desi importarile sunt executate la
    reincarcare, nu au efect, fiindca modulele B
    si C sunt deja importate
```

```
>>> import A
>>> # ...Modificare cod al lui A, B, C...
>>> from importlib import reload
>>> reload(A)  # Doar A este reincarcat
```

- Un reincarcator recursiv, *reloadall.py*, care foloseste atributul predefinit __*dict*__ si functia predefinita *type()* pentru reincarcarea tuturor modulelor:

```
#!python
"""
reloadall.py: reincarca tranzitiv modulele
    incluse (2.X + 3.X).
Functia reload_all are oricate argumente de tip
    obiect modul
"""
```

reloadall.py...



```
import types
import importlib
def status(module):
    print('reloading ' + module.__name__)

def tryreload(module):
    try:
        importlib.reload(module) # Uneori rateaza...
    except:
        print('FAILED: %s' % module)

def transitive_reload(module, visited):
    if not module in visited: # Fara duplicate
        status(module)
        tryreload(module)
        visited[module] = True
        for attrobj in module.__dict__.values(): # Toate atributele modulului
            if type(attrobj) == types.ModuleType:
                transitive_reload(attrobj, visited) # Apel recursiv daca de tip modul

def reload_all(*args):
    visited = {} # dict vid
    for arg in args: # Pentru fiecare argument
        if type(arg) == types.ModuleType:
            transitive_reload(arg, visited)
```

reloadall.py...

```
def tester(reloader, modname):  # Autotestare
    import sys
    if len(sys.argv) > 1:
        modname = sys.argv[1:]
    else:
        modname=[modname] # list!
    reloader(*(importlib.import_module(x) for x in modname)) # Suporta oricate module in linia de comanda, * despacheteaza toate obiectele modul ca argumente individuale

if __name__ == '__main__':
    tester(reload_all, 'reloadall') # Pe sine insusi
```

- Rulare:

```
C:\code>py -3 reloadall.py pybench sys string tkinter
reloading pybench
reloading sys
reloading string
reloading _string
...etc...
reloading copyreg
reloading tkinter
reloading _tkinter
reloading tkinter.constants
```

Reincarcare normala vs. tranzitiva

- Exemplu:

```
import b # Fisierul a.py
X = 1
import c # Fisierul b.py
Y = 2
Z = 3 # Fisierul c.py
C:\code> py −3
>>> import a
>>> a.X, a.b.Y, a.b.c.Z
(1, 2, 3)
>>> # Modificare X=111, Y=222, Z=333
>>> from importlib import reload
>>> reload(a) # Doar a este reincarcat
<module 'a' from
    'C:\\Users\\Dan\\Desktop\\code\\a.py'>
>>> a.X, a.b.Y, a.b.c.Z
(111, 2, 3)
>>> from reloadall import reload_all
>>> reload_all(a)
reloading a
reloading b
reloading c
>>> a.X, a.b.Y, a.b.c.Z # Toate reincarcate
(111, 222, 333)
```

reloadall2.py

- Versiune tot recursiva, dar cu *set* in loc de *dict* si mai putin cod:

```
"""
reloadall2.py: incarca tranzitiv modulele
    incluse (versiune alternativa)
"""
import types
from reloadall import status, tryreload, tester
    # Refoloseste cod din prima versiune

def transitive_reload(objects, visited):
    for obj in objects:
        if type(obj) == types.ModuleType
    and obj not in visited:
            status(obj)
            tryreload(obj)
            visited.add(obj)
            transitive_reload(obj.
    __dict__.values(), visited)

def reload_all(*args):
    transitive_reload(args, set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall2') # Autotestare
```

reloadall3.py

- <u>Versiune cu *list* care emuleaza o stiva – nerecursiva</u>:

```
"""
reloadall3.py: reincarca tranzitiv cu o stiva
"""
import types
from reloadall import status, tryreload, tester # Refolosire cod

def transitive_reload(modules, visited):
    while modules:
        next = modules.pop() # Sterge ultimul element din list
        status(next)
        tryreload(next)
        visited.add(next)
        modules.extend(x for x in next.__dict__.values() if type(x) == types.ModuleType and x not in visited) # Expresie generator, folosita de extend()

def reload_all(*modules):
    transitive_reload(list(modules), set()) # Tot set() pentru visited si se transforma tuple – modules - in list!

if __name__ == '__main__':
    tester(reload_all, 'reloadall3') # Autotestare
```

Rulari

- Desi ordinea reincarcarii modulelor poate sa difere, **setul** de module reincarcate este acelasi:

```
C:\Users\Dan\Desktop\code>pypy3
>>>> import os
>>>> res1 = os.popen('pypy3 reloadall.py tkinter').readlines()
>>>> res2 = os.popen('pypy3 reloadall2.py tkinter').readlines()
>>>> res3 = os.popen('pypy3 reloadall3.py tkinter').readlines()
>>>> res1 == res2, res1 == res3 # Liste de linii
(True, False)
>>>> set(res1) == set(res3) # Cu set() ordinea nu conteaza
True
```

- Observatie: numai instructiunile *import* pastreaza atribute de tip modul, *from* **nu!**

Sumar

- Concepte in proiectarea modulelor
- Mascarea datelor din module
- Caracteristici viitoare cu modulul __future__
- Utilizare mixta cu __name__ si __main__
- Exemplu de cod dual
- Modificarea lui sys.path
- Extensia *as* pentru import si from
- Exemplu, modulul ca obiect in Python
- Importarea cu nume ca string
- Reincarcarea tranzitiva a modulelor
- ***Erori in folosirea modulelor***

Conflicte de nume

- Module cu acelasi nume sunt selectate in ordinea din *sys.path*
- Nume identice pot fi evitate sau se pot folosi pachete de module cu cai unice
- Se poate evita mascarea modulelor standard cu un modul propriu, la fel numit, prin folosirea importarilor absolute din v3.x (**from __future__ import absolute_import** in v2.x). Astfel se sare peste pachetul propriu, iar importarea cu ***from .*** permite in continuare accesul la versiunea locala

Ordinea instructiunilor

- Instructiunile dintr-un modul se executa in ordinea in care sunt scrise:
  - Codul (de la inceput) **nu** poate referi nume asignate **mai jos** in fisier
  - Functiile sunt executate doar la apel, asa incat ele pot referi nume **oriunde** in fisier

```
func1() # Eroare, func1 nu exista inca
def func1():
    print(func2()) # OK, func2 este cautata mai tarziu, la apel!
```

```
func1() # Eroare, func2 nu a fost gasita inca
def func2():
    return "Hello"
func1() # OK, ambele exista acum
```

- Se recomanda amplasarea *def*-urilor la inceput, iar apoi restul codului, ca sa nu conteze ordinea apelurilor de functii

from doar copiaza nume

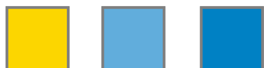
- *from* foloseste instructiunea de asignare:

```
# Fisier nested1.py:
X = 99
def printer(): print(X)

# Fisier nested2.py:
from nested1 import X, printer # Referinte
X = 88 # Acest X este local!
printer() # Vede X-ul sau, 99

% python nested2.py
99
```

- Cu *import*:

```
# Fisier nested3.py:
import nested1 # Obiect modul
nested1.X = 88 # Modificare atribut X din
    nested1
nested1.printer()

% python nested3.py # Rezultat corect
88
```

Cu *import* *, neclaritati de nume



- *import* * poate sa reasigneze variabile locale
- Sau este neclar de unde au fost importate

```
>>> from module1 import * # Variabile existente pot fi reasignate
>>> from module2 import * # Neprecizate...
>>> from module3 import *
>>> . . .
>>> func() # Care func?!
```

- Se recomanda:
  - listarea explicita a numelor importate
  - limitarea lui *import* * la un singur modul per fisier

Testare interactiva cu *reload()* si *from*

- Fie o sesiune interactiva:

from module import function	**function(1, 2, 3)**

- Dupa editare/eliminare bug:

from importlib import reload **reload(module)**	**function(1, 2, 3)** *# E vechea functie!*
from importlib import reload **import module**	**reload(module)** **function(1, 2, 3)** *# E tot vechea functie!!*
from importlib import reload **import module** **reload(module)**	**from module import function** *# Sau chiar module.function()* **function(1, 2, 3)** *# Functia **noua***

- Se mai poate lansa modulul modificat din **Idle**, cu (double)click, cu *exec(open('module.py').read())*

Importari circulare

- Situatia modulelor care se importa reciproc:

```
# Fisierul recur1.py:

X = 1

import recur2 # Modulul recur2 este creat acum

Y = 2
```

```
# Fisierul recur2.py:

from recur1 import X # OK, X exista acum in recur1

from recur1 import Y # Eroare, Y nu este inca creat/asignat
```

```
C:\code>py -3
>>> import recur1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\code\recur1.py", line 2, in <module>
    import recur2                         # Modulul recur2 este creat acum
  File "C:\code\recur2.py", line 2, in <module>
    from recur1 import Y                  # Eroare, Y nu este inca creat/asignat
ImportError: cannot import name 'Y' from 'recur1' (C:\code\recur1.py)
```

Importari...

- Se pot rezolva prin:
  - Proiectarea modulelor cu grad mare de coeziune si cuplaj minim
  - Amanarea accesului la atributele modulului cu *import* in loc de *from*
  - Executarea instructiunii *from* cat mai tarziu:
    - din interiorul unei functii – deci la apelul functiei
    - cat mai jos in modul